Scientific Workflows for Hadoop

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von

Dipl.-Inf. Marc Nicolas Bux

Präsidentin der Humboldt-Universität zu Berlin:
Prof. Dr.-Ing. habil. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter:

1. Prof. Dr.-Ing. Ulf Leser
2. Prof. Dr. Björn Scheuermann
3. Prof. Dr.-Ing. habil. Bernhard Mitschang

Tag der mündlichen Prüfung: 18. Juli 2018

Zusammenfassung

Scientific Workflows bieten flexible Möglichkeiten für die Modellierung und den Austausch zunehmend komplexer werdender Arbeitsabläufe zur Analyse wissenschaftlicher Daten. In den letzten Jahrzehnten sind verschiedene Scientific-Workflow-Management-Systeme entstanden, die den Entwurf, die Ausführung und die Verwaltung solcher Scientific Workflows unterstützen und erleichtern. In mehreren wissenschaftlichen Disziplinen wachsen die Mengen zu verarbeitender Daten inzwischen jedoch schneller als die Rechenleistung und der Speicherplatz der zur Verarbeitung dieser Daten verfügbaren Maschinen. Dies gilt insbesondere für die Lebenswissenschaften, in denen neue Technologien den Durchsatz der Sequenzierung genomischer Daten von einigen Kilobytes auf mehrere Terabytes pro Tag angehoben haben.

Parallelisierung und verteilte Ausführung werden typischerweise angewendet, um mit derart wachsenden Datenmengen Schritt zu halten. Allerdings ist eine Parallelisierung von Scientific Workflows in vielen Fällen schwierig umzusetzen. Außerdem sind die durch groß angelegte, verteilte Infrastrukturen bereitgestellten Rechenressourcen häufig heterogen, instabil und unzuverlässig. Um die theoretische Skalierbarkeit solcher Infrastrukturen dennoch uneingeschränkt nutzen zu können, müssen sich Scientific-Workflow-Management-Systeme weiterentwickeln: Potentiale für die Parallelisierung von Scientific Workflows müssen erfolgreich erkannt und ausgenutzt werden, um eine Verteilung der zugrundeliegenden Arbeitslast zu ermöglichen. Simulations-Frameworks, welche häufig zur Evaluation verteilter Planungsalgorithmen eingesetzt werden, müssen die Instabilität und Variabilität verteilter Infrastrukturen berücksichtigen. Adaptive Planungsalgorithmen müssen verwendet werden, um die Nutzung instabiler und in Ihrer Leistung veränderlicher Ressourcen zu optimieren. Moderne Systeme zur skalierbaren Verwaltung verteilter Rechen- und Speicherplatzressourcen, wie Apache Hadoop, müssen eingesetzt werden.

Diese Dissertation präsentiert neuartige Lösungsansätze für all diese Anforderungen. Zunächst stellen wir DynamicCloudSim vor, ein Simulations-Framework für Cloud-Infrastrukturen, welches dazu in der Lage ist, die verschiedenen Aspekte der Variabilität solcher Infrastrukturen adäquat zu modellieren. Im Anschluss beschreiben wir ERA, einen adaptiven Planungsalgorithmus, der die Ausführungszeit eines Scientific Workflows optimiert, indem er Heterogenität ausnutzt, kritische Teile des Scientific Workflows repliziert und sich an Veränderungen in der zugrundeliegenden Infrastruktur anpasst. Schließlich präsentieren wir Hi-WAY, eine Ausführungsumgebung, die ERA integriert, und die hochgradig skalierbare Ausführungen in verschiedenen Sprachen beschriebener Scientific Workflows auf Hadoop ermöglicht.

Abstract

Scientific workflows provide a flexible means to model, execute, and exchange the increasingly complex analysis pipelines necessary for today's data-driven science. Over the last decades, scientific workflow management systems have emerged to facilitate the design, execution, and monitoring of such workflows. At the same time, the amounts of data generated in various areas of science outpaced advancements in computational power and storage capabilities. This is especially true for the life sciences, where new technologies increased the sequencing throughput from kilobytes to terabytes per day.

Parallelization and distributed execution are generally proposed to deal with these increasing amounts of data. However, parallelization of scientific workflows is, in many cases, difficult to realize. Also, the computational resources provided by large-scale, distributed infrastructures are subject to heterogeneity, dynamic performance changes at runtime, and occasional failures. To leverage the theoretical scalability provided by these infrastructures despite the observed aspects of performance variability, workflow management systems and the tools backing their development have to progress: Parallelization potentials in scientific workflows have to be detected and exploited to allow for a distribution of workload. Simulation frameworks, which are commonly employed for the evaluation of distributed scheduling mechanisms, have to consider the instability encountered on the infrastructures they emulate. Adaptive scheduling mechanisms have to be employed to optimize resource utilization in the face of instability. State-of-the-art systems for scalable distributed resource management and storage, such as Apache Hadoop, have to be supported.

This dissertation presents novel solutions for these aspirations. First, we introduce DynamicCloudSim, a cloud computing simulation framework that is able to adequately model the various aspects of variability encountered in computational clouds. Secondly, we outline ERA, an adaptive scheduling policy that optimizes workflow makespan by exploiting heterogeneity, replicating bottlenecks in workflow execution, and adapting to changes in the underlying infrastructure. Finally, we present Hi-WAY, an execution engine that integrates ERA and enables the highly scalable execution of scientific workflows written in a number of languages on Hadoop.

Acknowledgments

I am deeply thankful to a number of people without whom this dissertation thesis would not have been possible. First and foremost, I want to express my sincere gratitude to my supervisor Prof. Ulf Leser. He gave me all the freedom to explore, experiment, and learn, while providing me with counseling and guidance when needed. He was always positive, considerate, and fair, and gave invaluable contributions and extensive feedback to all my scientific works. Secondly, I want to thank my second supervisor Prof. Björn Scheuermann for a series of very constructive brainstorming sessions during which we formalized the ERA scheduler over cups of black tea. Thirdly, I would like to thank Assoc. Prof. Jim Dowling for introducing me to Hadoop YARN and letting me join his group for a Swedish summer, in which the foundations for Hi-WAY were laid. Finally, I am grateful to the graduate school SOAMED for providing an environment in which to strive through soft-skill courses and biyearly retreats.

I am also grateful to my colleagues of the WBI group for the wonderful discussions, collaborations, dinners, joint conference visits, board game nights, road trips, hiking vacations, cakes, and encouragements. I would like to thank Philippe Thomas for supervising my diploma thesis, hiking through the grand European wilderness with me, and always helping me out with invaluable advice. I am grateful to Patrick Schäfer for our epic board game nights as well as for co-organizing courses from which I learned at least as much as the students. I would like to thank Carl Witt for always lightening up the mood of people he surrounds himself with as well as for continuing the Hi-WAY project. I am also grateful to Jörgen Brandt for our extensive collaboration over the years, André Koschmieder for all the fun we had skiing, hiking, and conversing, Sebastian Wandelt who encouraged me to pursue my doctoral studies, Karin Zimmermann for finding me the most beautiful place to live, Astrid Rheinländer for always being true, direct, and raising the bar in science, Tim Rocktäschel, Liam Childs, Yvonne Lichtblau, and Stefan Kröger for countless insightful discussions and the wonderful time spent together, as well as Carsten Lipka, Hannes Schuh, and Christopher Schiefer for fruitful collaborations.

On a personal note, I am very grateful to my parents, who, through leading by example (and early exposure to Star Trek) raised me with an unwavering optimism – a quality I found very helpful during my doctoral studies. Also, I would like to thank my friends from close and afar, who always challenge my way of thinking and remind me of what life is all about. Finally, I am deeply grateful to my girlfriend Luise Bernhardt who I value more than anything or anyone else and who, as a consequence of our scientific endeavors, had to travel back and forth between Berlin and Erlangen for four years.

I am also very thankful for the funding I received through the DFG graduate school SOAMED, the EU project BiobankCloud, the German Academic Exchange Service, and an AWS in Education Research Grant award.

Contents

1 Introduction

Over the last decades, computation has been established as an integral part of research (Hey et al., 2009). Today's scientific experiments typically involve running chains of computational analysis tasks to synthesize succinct results by transforming, filtering, and aggregating large amounts of data. The tasks used within these data analysis pipelines are created by thousands of researchers around the world, rely on domain-specific data exchange formats, and are updated frequently (Pabinger et al., 2014; Momcheva and Tollerud, 2015). The programming model of scientific workflows facilitates the implementation, execution, maintenance, and exchange of such analysis pipelines (Deelman et al., 2009).

Recent years have brought an unprecedented influx of data across many fields of science (Stephens et al., 2015). In fact, the amount of data produced in many scientific domains has risen at exponential rates and often outpaced advances in storage capacity, network bandwidth, and processing power (Kahn, 2011). In genomics, for instance, the latest generation of genomic sequencing machines can handle up to 18,000 human genomes per year (Van Dijk et al., 2014), generating about 30 to 50 terabytes of sequence data per week. Consequently, the computational cost of running certain scientific workflows is becoming increasingly difficult for a single machine to handle. This dissertation studies the scalable execution of such data-intensive scientific workflows.

Besides algorithmic advances, the canonical way to deal with increasing data volumes is parallelization and distribution of workload across multiple processing cores of a single computer or nodes of computer clusters, grids, and clouds. Infrastructure-as-a-service clouds such as Amazon's Elastic Compute Cloud (EC2) are particularly well-suited to keep up with the generation of scientific data, since they provide arbitrarily scalable compute and storage resources on demand (Stein, 2010). However, computational clouds are subject to a considerable degree of heterogeneity (Dejun et al., 2009; Schad et al., 2010), unpredictable performance changes at runtime (Zaharia et al., 2008), and occasional straggler, unresponsive, or otherwise faulty machines (Jackson et al., 2010). These aspects of performance variability have to be accounted for when scheduling and executing scientific workflows.

Distributed resource management systems like Hadoop YARN (Vavilapalli et al., 2013) or Mesos (Hindman et al., 2011) have been developed to employ thousands of compute nodes in parallel and are therefore able to fully leverage the theoretical scalability offered by cloud infrastructures. Furthermore, they provide the transparency and fine-grained control over resources required for adaptive scheduling policies that provide robustness to or even exploit the instability and heterogeneity inherent to computational clouds and similar distributed infrastructures. Hadoop also provides an implementation of the MapReduce programming model (Dean and Ghemawat, 2008) along with

its own distributed file system HDFS, which tolerates the failures of storage nodes and distributes network load by storing data redundantly. For these reasons, Hadoop has found widespread adoption both in academia and in the industry.

Unfortunately, established scientific workflow management systems do not support modern distributed resource management systems like Hadoop. In addition, they also disregard the speed-ups and increased robustness to be gained from adaptive workflow scheduling policies. Instead, they either employ static scheduling policies, in which task-machine assignments are computed ahead of execution, or implement some naive form of online scheduling, which is entirely oblivious to the performance requirements of a workflow's tasks as well as the capabilities of the computational infrastructure (Bux and Leser, 2013b).

The previous paragraphs highlighted a gap between the increasing computational cost of executing data-intensive scientific workflows and the capabilities of established workflow management systems with regard to distributed execution and adaptive scheduling. As a consequence, a number of works have been published on the re-implementation of popular scientific workflows and the tasks they comprise as highly parallelizable sequences of MapReduce programs, to be executed on a scalable Hadoop installation (Decap et al., 2015; Zou et al., 2013). However, to date there has been no published work on natively supporting Hadoop YARN, or any comparable distributed resource management system, as a scientific workflow management system's primary processing engine.

1.1 Contributions

This dissertation thesis covers the scalable execution of data-intensive scientific workflows on infrastructure-as-a-service clouds and similar distributed infrastructures shared between multiple users. Within this topic, it focuses on the problem of adaptive workflow scheduling, i. e., adapting workflow execution to the dynamic performance variations and instability encountered on such infrastructures. To this end, it gives an overview of the state of the art in scalable workflow execution, presents a workflow simulation toolkit able to model performance variations and proposes a novel adaptive workflow scheduling scheme. Finally, it presents a highly scalable scientific workflow management system that implements this scheduling scheme and builds on top of the well-established distributed resource management system Hadoop YARN.

The specific contributions of this dissertation thesis are as follows:

1. To outline the design choices for the scalable execution of scientific workflows, we present a light-weight taxonomy that encompasses the following key concepts: (i) computational infrastructures, (ii) distributed processing frameworks, (iii) adaptivity in scheduling, and (iv) parallelization techniques. We illustrate these concepts using a number of popular workflows from the fields of computational genomics and astronomy. Furthermore, we give an exhaustive overview of current workflow management systems' capabilities with regard to the four aforementioned concepts.

2. A particular problem in the development and systematic evaluation of novel adaptive workflow scheduling policies is the substantial monetary cost required for repeatedly setting up and executing long-running computational experiments. As an alternative, simulation provides a quick, affordable, and reproducible means of assessment and is therefore commonly employed in the evaluation of novel scientific workflow scheduling policies (Blythe et al., 2005). We present DynamicCloudSim, an extension to the established simulation toolkit CloudSim. DynamicCloudSim augments CloudSim with a fine-grained representation of computational resources, the capability to simulate the execution of arbitrary scientific workflows, and models for mimicking several aspects of variability, including node heterogeneity, dynamic changes of performance at runtime, and failures during task execution.

3. Aspects of performance variability inherent to shared, distributed infrastructures such as infrastructure-as-a-service clouds, have to be accounted for when scheduling and executing scientific workflows. An adaptive workflow scheduler provides robustness to instability and even exploits heterogeneity by monitoring workflow execution and by dynamically adapting to alterations in the underlying computational infrastructure. We introduce ERA, a scientific workflow scheduling policy that achieves adaptivity by (i) exploiting heterogeneity in to-be-scheduled tasks and available resources, (ii) replicating bottleneck tasks running on subpar machines during workflow execution, and (iii) adapting to unforeseeable performance changes by means of a stochastic performance model learned online during workflow execution. Simulation experiments in DynamicCloudSim indicate significant reductions in workflow makespans over established workflow schedulers.

4. Established scientific workflow management systems do not support modern distributed resource management systems like Hadoop YARN and neglect adaptivity in workflow scheduling. We present the scientific workflow execution engine Hi-WAY, which is part of the scientific workflow management system SAASFEE. Hi-WAY is able to execute scientific workflows specified in a multitude of different languages on Hadoop YARN, harnessing its proven scalability. It optimizes performance for different utilization scenarios by implementing a number of scheduling policies, including ERA. It achieves reproducibility of workflow executions through (i) the automated setup of infrastructures and (ii) support for re-executable provenance traces. We demonstrate Hi-WAY's key properties of scalability and performance by several case studies.

1.2 Thesis Outline

Chapter 2 gives a comprehensive overview of the state-of-the-art in implementing and executing data-intensive scientific workflows. We commence by formally introducing key concepts and by describing real-life scientific workflows from the areas of computational genomics and astronomy, which shall be used repeatedly throughout this thesis. We provide a taxonomy of design choices critical to the implementation and exploitation

of parallelization in such workflows, i. e., basic parallelization techniques, computational infrastructures, distributed processing frameworks, and adaptivity in scheduling. Subsequently, we categorize established scientific workflow management systems based on whether and how they realize these concepts. Finally, we outline how deficiencies of available scientific workflow management systems have led to the emergence of several makeshift solutions for specific workflows and domains.

In Chapter 3 we describe the simulation framework DynamicCloudSim, which enables adequate simulations of scientific workflow enactment on cloud computing infrastructure. The models of variability implemented in DynamicCloudSim are based on empirical analyses on the performance of Amazon EC2, which constitutes the largest and most well-studied commercial cloud. As a validation of DynamicCloudSim's functionality, we simulate the impact of instability on scientific workflow scheduling by assessing and comparing the performance of four workflow schedulers in the course of several experiments both in simulation and on real cloud infrastructure. Results indicate that our model is able to adequately capture the most important aspects of cloud performance variability.

Chapter 4 presents ERA, a novel policy for the adaptive scheduling of scientific workflows on large distributed infrastructures subject to instability. ERA employs heuristics to exploit heterogeneity, adapt to performance changes, and cope with bottlenecks during workflow execution. To identify favorable assignments of tasks to machines, ERA obtains task runtime estimates from historical runtime measurements. To this end, it models a task's performance on a given machine as a stochastic Wiener process. We determine suitable parameters for ERA and perform an evaluation in DynamicCloudSim simulating different workflows and infrastructures with varying degrees of heterogeneity and instability.

In Chapter 5, we present the scientific workflow execution engine Hi-WAY, which enables the scalable execution of adaptively scheduled scientific workflows written in various languages on top of Hadoop YARN. We describe the architecture of Hi-WAY and highlight its most important features: performance gains through adaptive scheduling, scalability, support for multi-language and iterative workflows, and reproducibility of experiments. Subsequently, we report on several experiments, in which workflows from different scientific domains were repeatedly executed on local clusters as well as on virtual clusters in Amazon EC2 comprising up to 128 worker nodes. We also evaluate the benefits of adaptive scheduling and the performance of the ERA scheduler on real cloud infrastructure.

Finally, we give a summary of this thesis along with an outlook on arising research questions in Chapter 6.

1.3 Own Prior Work

Some contents of this thesis have been published previously.

Wandelt et al. (2012) outlined current developments in the storage and processing of next-generation sequencing data. In this publication, Rheinländer and Thalheim gave an overview of standalone and cloud-based read mapping tools. Wandelt reviewed how

traditional compression algorithms could be applied to sequencing data and outlined the benefits to be gained from applying referential compression schemes instead. Bux gave a general characterization of analysis pipelines in next-generation sequencing, excerpts of which can be found in Section 2.1.1 of this thesis. Leser structured and directed the joint work, wrote both the introduction as well as the conclusion, and, together with Haldemann, reviewed the manuscript.

Bux and Leser (2013b) gave an in-depth review on parallelism in scientific workflows and their implementation in current workflow management systems. Bux conducted the literature research, devised the presented taxonomy, characterized scientific workflow management systems available at the time according to this taxonomy, and wrote the manuscript, parts of which can be found in Chapters 1 and 2. Leser supervised the work and reviewed the document.

Bux and Leser (2013a) presented DynamicCloudSim, an extension to the cloud simulation toolkit CloudSim, that introduces models for different aspects of variability. Bux programmed DynamicCloudSim, conducted the simulation experiments, and wrote the manuscript. Bux and Leser (2014) later published an extended version of the article, in which additional validation experiments on real hardware were described. Bux conducted these experiments and wrote the extended manuscript, parts of which can be found in Chapter 3 of this thesis. Leser supervised the work and reviewed the text.

Bux et al. (2017) published Hi-WAY, an execution engine enabling the scalable execution of scientific workflows written in various languages on the distributed processing framework Hadoop. Bux programmed Hi-WAY, conducted a range of experiments in which different properties of Hi-WAY were evaluated, and wrote the manuscript. Parts of this manuscript can be found in Chapter 5. Leser and Dowling supervised the work on Hi-WAY. Leser, Brandt, and Witt reviewed the document.

Bux et al. (2015) provided a description and demonstration of the scientific workflow management system SAASFEE, bundling the workflow language Cuneiform, the Hi-WAY execution engine, and the distributed processing framework Hadoop. Brandt wrote the text passage on Cuneiform, whereas Bux wrote the text on Hi-WAY. The section on demonstration workflows as well as the introduction and conclusion were written jointly and in equal parts by Bux and Brandt. The experiment in which a workflow was executed using both Hi-WAY and Apache Tez was conducted by Lipka and Bux. A description of SAASFEE as well as the results of the aforementioned experiment can be found in Figure 5.1 and Section 5.2.1, respectively. Leser supervised the work and reviewed the article.

Bessani et al. (2015) gave an overview of the BiobankCloud platform-as-a-Service for the secure storage and processing of next-generation sequencing data. Marking the conclusion of an EU project that had been running for three years, a total of 17 people worked on the document. Bux and Brandt wrote the section on SAASFEE. Leser and Dowling supervised the work and reviewed the document.

2 Data-Intensive Scientific Workflows

Scientific workflows have recently emerged as a flexible programming model for processing scientific data (Deelman et al., 2009). However, increasing amounts of scientific data have eventuated in ever-growing requirements of computational power and an elevated demand for parallelization and distributed execution of scientific workflows. This chapter outlines the formalisms behind, requirements for, and existing approaches towards realizing the scalable execution of scientific workflows on distributed computational infrastructures.

In Section 2.1, we formally introduce the programming model of scientific workflows. In Sections 2.1.1 and 2.1.2, we present three distinct data-intensive scientific workflows from the fields of computational genomics and astronomy. These workflows are used as a reference and evaluation baseline throughout this thesis. They were selected since they are well-studied examples from different scientific domains. Also, they have been repeatedly utilized for evaluating aspects of scientific workflow execution and, most notably, scheduling in the past (Juve et al., 2012). They can be scaled to nearly any desired size and degree of parallelism by adjusting the amount of to-be-processed input data.

Subsequently, we outline fundamental requirements for the scalable execution of scientific workflows in Section 2.2. This encompasses parallelization techniques available for scientific workflows, distributed computational infrastructures and processing frameworks, as well as adaptivity in workflow scheduling. All of these requirements build on top of one another and we shall argue that adaptive scheduling heuristics are key to unlock the scalability provided by today's distributed architectures.

In Section 2.3, we review established scientific workflow management systems. We showcase how some systems neglect scalable workflow execution altogether, focusing on other aspects of workflow management instead, while others attempt to meet at least some of the aforementioned requirements.

We continue in Section 2.4 by outlining how, in the field of computational genomics, the neglect of scientific workflow management systems to fully embrace scalability and parallelization, has prompted the development of various makeshift solutions. These solutions include both re-implementations of domain-specific tools and libraries, as well as the outsourcing of computationally demanding tasks within a workflow.

We then conclude by summarizing the state of data-intensive scientific workflow management in Section 2.5.

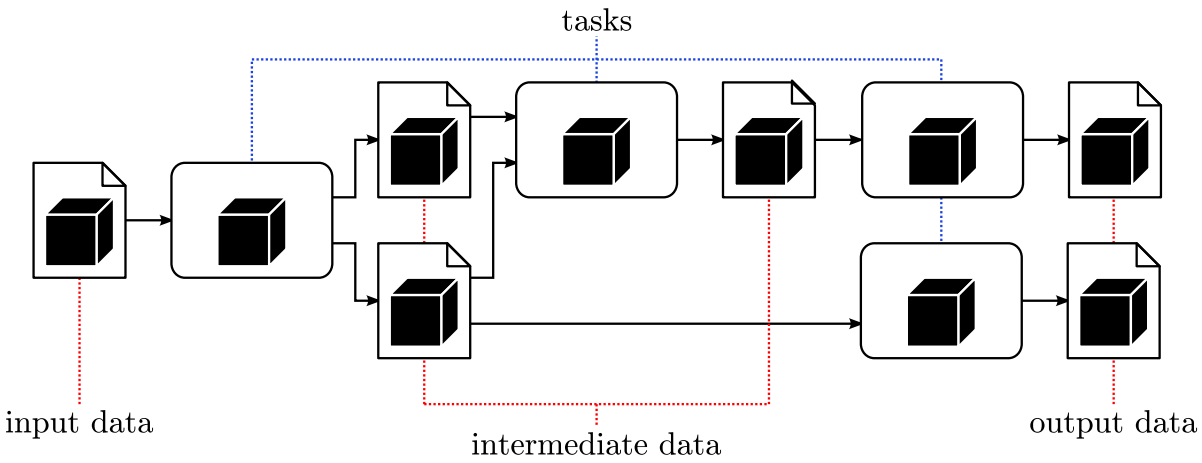


Figure 2.1: An exemplary scientific workflow. Input data is transformed into output data by invoking a number of sequential and concurrent tasks. Usually, all involved data are materialized as files. Tasks as well as the data they process and produce are treated as black boxes.

2.1 Scientific Workflows

The programming model of scientific workflows enables the modeling and automated execution of data analysis pipelines typically encountered in today's scientific research (Deelman et al., 2017). A scientific workflow is composed of sequential and concurrent data processing tasks, whose order is determined by data dependencies (Taylor et al., 2007). See Figure 2.1 for a visualization of a scientific workflow.

Definition 1 (Scientific Workflow) *A scientific workflow is a bipartite, directed, and acyclic graph $S = (D, T, E)$ comprising a set of data and task vertices D, respectively T, and a set of edges $E \subseteq (D \times T) \cup (T \times D)$.*

Note that definitions of scientific workflows exist in which data is modeled only implicitly in the form of precedence constraints represented as edges between tasks in a graph or multigraph. However, this definition does not allow a one-to-one mapping between data and edges, since data is produced by at most one task, yet can be consumed by several upstream tasks (consider, for instance, the scientific workflow displayed in Figure 2.1).

A scientific workflow's data $d \in D$ are usually materialized as files on a local or remote hard disk. These data are treated as black boxes and can be structured or unstructured, binary or text-based, etc. Any data that has to be present for the workflow to be run, since it is not produced by any task, is called *input* data. Similarly, data that is produced by a task is called *intermediate* or *output* data, depending on whether it is processed by another task or not.

Similar to the data they process and produce, tasks $t \in T$ are are also treated as black boxes: They can be written in any programming language, may perform computations locally, interface with external databases, invoke remote web services, etc.

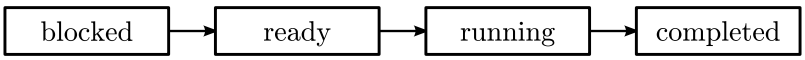


Figure 2.2: The life cycle of a task during scientific workflow execution. A task is blocked until all of its data dependencies are resolved. It is then ready for execution and, upon assignment to a machine, will be running until it is completed.

Definition 2 (Task) *A task is the smallest divisible unit of work in a scientific workflow. It invokes a non-interactive, executable computer program and, as a result of computation, generates output data. It may also read input data which influences the results of computation.*

A task can undergo several stages during workflow execution, as displayed in Figure 2.2. Until all of its input data are available it is *blocked* and awaiting the completion of all of its preceding tasks. As soon as these parent tasks have completed, the task is *ready* for execution. During some point of workflow execution, it is *running* on a machine until it is *completed*.

Two tasks are called to belong to the same *bag* if they invoke the same program, possibly for different input data. A *bag of tasks* within a given scientific workflow is a set of tasks belonging to the same bag (Cai et al., 2017). Data-intensive scientific workflows can easily comprise several bags of thousands of tasks each (see, for instance, the workflows in Sections 2.1.1 and 2.1.2).

Definition 3 (Bag-of-Tasks Workflow) *A bag-of-tasks workflow is a pair* $(S, \mathcal{B})$*, where* $S = (D, T, E)$ *is a scientific workflow and* $\mathcal{B} = \{B_0, B_1, ..., B_p\}$ *is a set of bags of tasks that partitions* T*.*

To facilitate the modeling and generate succinct graphical representations of such bag-of-tasks workflows, abstraction is a commonly used strategy. An *abstract workflow* is a graphical representation of a bag-of-tasks workflow or group of bag-of-tasks workflows, in which bags of tasks are merged into composite vertices representing conceptual processing steps. In such abstract workflows, data vertices are either merged into composite vertices as well, or omitted altogether. Scientific workflows discussed in the remaining parts of this section are displayed as abstract workflows (see Figures 2.4 to 2.6).

2.1.1 Scientific Workflows for High-Throughput Genomics

Genomic sequencing denotes the process that determines the sequence of nucleotides within a given DNA molecule. For long molecules such as chromosomes, this can so far only be achieved by splitting the DNA into many short, overlapping fragments, which are called reads (Shendure and Ji, 2008). High-throughput sequencing (also known as second generation or next-generation sequencing) technology has given biologists and clinicians insights into masses of individual genomes at reasonable speed and affordable cost (Van Dijk et al., 2014). Due to the diversity of emerging research questions, hundreds of tools have been developed and are employed for the analysis of next-generation sequencing

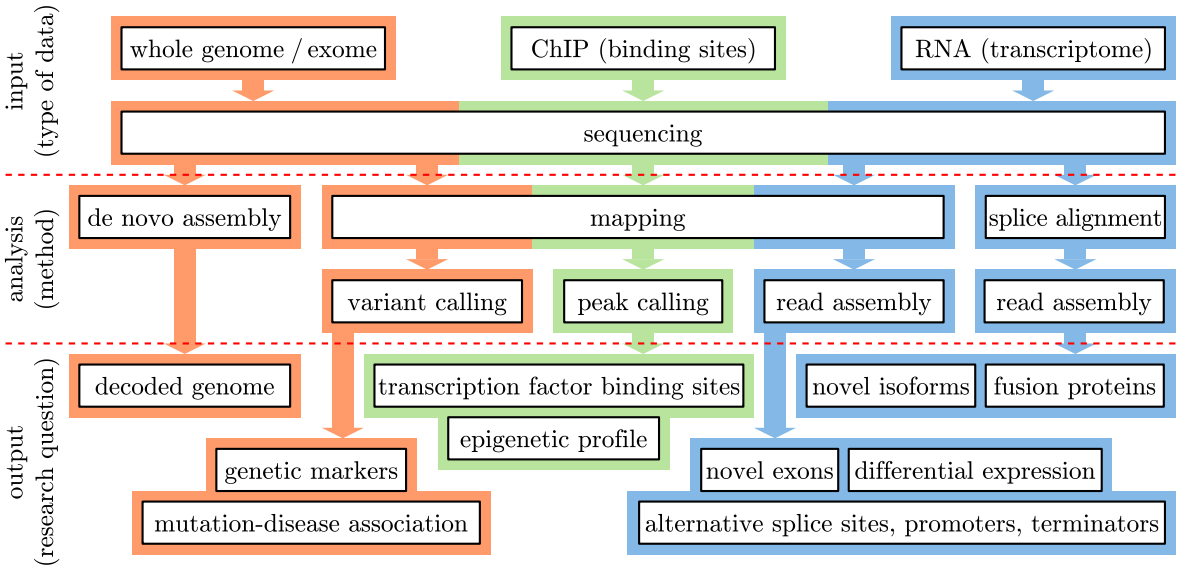


Figure 2.3: A selection of typical analysis pipelines for processing high-throughput sequencing data. Depending on the type of input data, i. e., whole genome sequencing, ChIP-seq, or RNA-seq data, different analysis methods are available to obtain output data for answering various research questions.

data (Pabinger et al., 2014). These tools are usually used in combination with one another, forming complex and intertwined analysis pipelines, which can be implemented as scientific workflows. The composition of such analysis pipelines depends on the type (i. e., cellular origin) of sequencing data along with the nature of the research question at hand (see Figure 2.3). In this section, we dissect the typical structure of analysis pipelines processing high-throughput sequencing data.

The primary output generated by high-throughput sequencing machines is usually available in textual FASTQ format. FASTQ files comprise sets of DNA reads, each with their respective identifier, base sequence and quality scores for each base call. Quality scores are mostly utilized to assess and filter low-quality reads prior to further analysis.

Reconstructing the underlying genome from the remaining short reads constitutes the first major step of most analysis pipelines. The way this genome reassembly is approached largely depends on whether a closely related genome of the same species is available as reference. If a reference genome is present, reads can be mapped to this reference. Otherwise, the genome has to be assembled without exterior knowledge. Both techniques – a *de novo* assembly even more so than a reference mapping – are prone to error and computationally demanding. A plethora of tools has been developed for both approaches, yet no standards have been established.

Further steps in the analysis pipeline mostly depend on the research question at hand. Subsequent to reference mapping, the detection of genetic variants is a common goal. A variant denotes a base that is different between the reference and the newly sequenced read, hinting towards a mutation with potential consequences for the organism (Pabinger et al., 2014). Besides these single nucleotide variants, smaller insertions, deletions, or

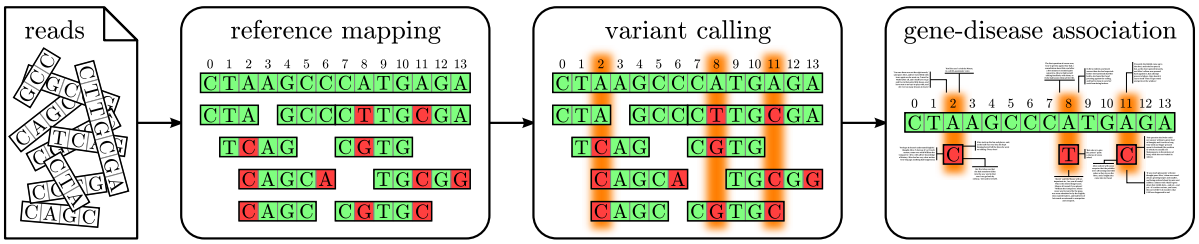


Figure 2.4: An abstract SNV calling workflow. First, reads obtained via high-throughput sequencing of pathogenic tissue are mapped to a reference genome. Secondly, the mapped reads are compared against the reference to detect variants, which serve as indicators for mutations. Finally, determined variants are filtered and functionally characterized. Mutations specific to the sequenced sample might be functionally associated to the investigated disease.

larger structural variants are also important. Identified variants undergo quality assessment, filtering and characterization of functionality and specificity. This aggregated information can, for instance, be utilized to determine associations between a disease and mutations (Li and Leal, 2008). In Section 2.1.1.1 we present a scientific workflow that comprises a reference mapping, variant calling, and gene-disease association.

Other common research questions include (i) the discovery of differential expressions or splice variants in transcriptome sequences (RNA-seq), (ii) the determination of bindings of proteins to DNA to elucidate regulatory relationships between genes and transcription factors, or (iii) the study of evolutionary relationships between species or individuals. All these problems boil down to a series of computationally demanding algorithms operating on genomic sequences or derived information. Section 2.1.1.2 outlines a workflow that determines the differential expression of transcriptomes based on RNA-seq data.

Independent of the research question at hand, ever-growing volumes of data generated by high-throughput sequencing machines increasingly necessitate the distribution of storage and computation and, in particular, the employment of cloud computing technology (Pennisi, 2011). This ongoing development constitutes a driving motive behind core topics of this thesis, i. e., the implementation of data-intensive analysis pipelines as parallelizable scientific workflows as well as the scalable and robust execution of these workflows on shared distributed infrastructures.

2.1.1.1 Single Nucleotide Variant Calling

Among the research questions outlined in Section 2.1.1, single nucleotide variant (SNV) calling is one of the most well-studied and oft-encountered. Consequently, different implementations of SNV calling workflows are employed for illustration and evaluation purposes several times within this thesis (see Sections 2.2.1, 3.6, 4.3.5, 4.3.6, and 5.2.1). These workflows process genomic reads, usually obtained from sequencing the genome or exome of pathogenic (disease) tissue, and can be utilized to gain insights or ascertain hypotheses regarding the association of mutations on genes with known diseases. Figure 2.4 provides an abstract illustration of this group of workflows.

In the first step of a SNV calling workflow, which is referred to as reference mapping, genomic reads are mapped against an established (and different) reference to determine their original position in the genome. The landscape of available reference mapping tools is vast; a comprehensive survey on mapping tools was given by Li and Homer (2010). Tools used in experiments discussed in this thesis include Bowtie (Langmead et al., 2009b), Bowtie 2 (Langmead and Salzberg, 2012), BWA (Li and Durbin, 2009), PerM (Chen et al., 2009), and SHRiMP (David et al., 2011). Different mapping tools produce different results. Therefore, a common technique to increase overall mapping quality involves running several tools either in parallel, combining the results in a subsequent step, or in sequence, re-mapping reads that failed to map using a different tool.

Subsequent to reference mapping, the mapped reads are piled up and each position is compared to the reference genome to detect variants, i. e., mismatching nucleic acids, which might be indicative of mutations. Again, the amount of available tools is considerable; a comprehensive survey of tools for variant calling and analysis has been presented by Pabinger et al. (2014). Finally, characteristics of detected variants such as rarity or functional importance are obtained from external databases like dbSNP (Sherry et al., 2001) or others (Childs et al., 2016). In experiments described in this thesis, variants are determined from mapped reads using SAMtools (Li et al., 2009), VarScan (Koboldt et al., 2009), and MuTect (Cibulskis et al., 2013); variant characterization is performed using the tool ANNOVAR (Wang et al., 2010).

Characterized variants can be investigated closer or compared to variants determined from processing other genomic samples using the same workflow. Mutations specific to the pathogenic genotype might be related to the disease. The associated gene could therefore qualify as a drug target (Li and Leal, 2008).

Numerous implementations of SNV calling workflows as well as extensions and parts thereof have been developed and compared against one another (Cornish and Guda, 2015). Implementations of such workflows have also been provided as components of genome analysis frameworks such as GATK (McKenna et al., 2010) or ADAM (Nothaft et al., 2015).

2.1.1.2 RNA Sequencing

RNA sequencing (RNA-seq) technology makes use of high-throughput sequencing to enable researchers to determine and quantify the transcription level of genes in a given tissue sample. Trapnell et al. (2012) have developed a workflow that has been established as the *de facto* standard for processing and comparing RNA-seq data. It can be employed to identify new genes and splice variants and to determine differential transcription of genes between samples, which, in turn, can help to understand the misregulation of gene expression in disease. In the context of this thesis, this workflow has been employed for evaluation in Section 5.2.2. See Figure 2.5 for a visualization of this workflow.

The input to this workflow are genomic reads obtained by sequencing the transcriptome, i. e., the set of transcribed genes. In the first step of the workflow, these reads are mapped against a reference genome using the two mapping tools Bowtie 2 (Langmead and Salzberg, 2012) and TopHat 2 (Kim et al., 2013). Similar to the mapping step of

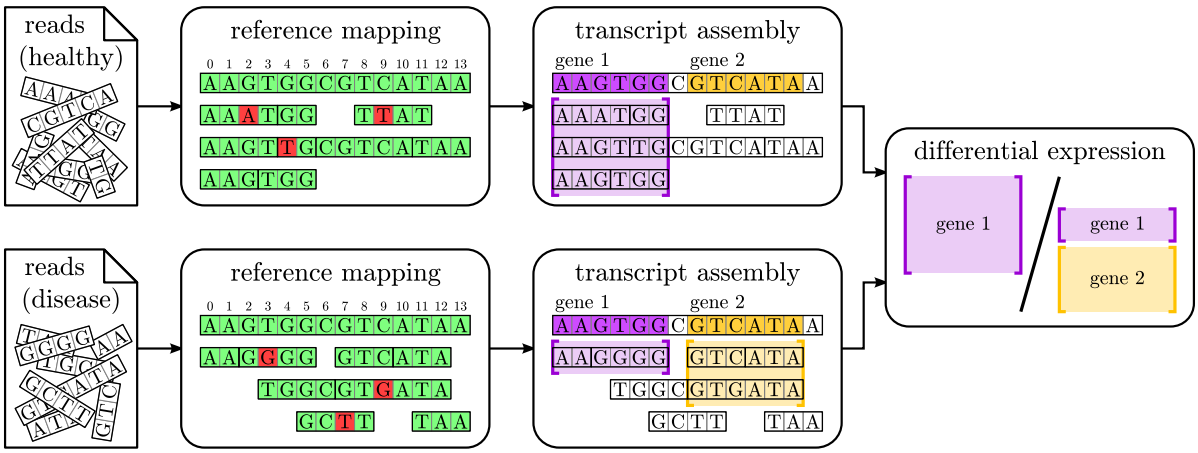


Figure 2.5: An abstract RNA sequencing workflow for determining differential transcription. Genomic reads, the output of whole transcriptome sequencing (RNA-seq), are mapped against a reference genome. Transcribed genes are then determined and quantified based on these mappings. Finally, transcription is compared between samples.

the SNV calling workflow presented in Section 2.1.1.1, this step serves the purpose of identifying the reads' genomic positions, which have been lost during the sequencing process.

In the second step, the Cufflinks (Trapnell et al., 2012) package is utilized to assemble and quantify transcripts of genes from mapped reads. Quantified transcripts are then compared between different input samples, for instance between pathogenic and healthy or between pathogenic and medicated samples. The resulting output of the workflow – differential transcription of genes between samples – can shed light into alterations of gene transcription as a consequence of disease or treatment, and thus reveal new treatment options.

2.1.2 Montage: Astronomical Mosaics of the Sky

As outlined in Section 2.1, the tasks composing a scientific workflows and the data they process and produce are viewed as black boxes. The techniques described in this thesis are therefore not confined to the field of high-throughput genomics, but can be applied to any scientific domain. Here, we outline the group of Montage workflows from the field of astronomy, which have often been used for evaluating scientific workflow scheduling and execution (Deelman et al., 2008; Hoffa et al., 2008; Lee et al., 2009; Chen and Deelman, 2012).

The Montage toolkit comprises a set of applications enabling the assembly of high-resolution mosaics of regions of the sky (Berriman et al., 2004). Montage is able to automatically generate workflows in DAX format (see Section 2.3.1.1) for building mosaics of configurable size. The size of a region of the sky is typically measured in square degree units, with one square degree approximately covering the area of five full moons. When

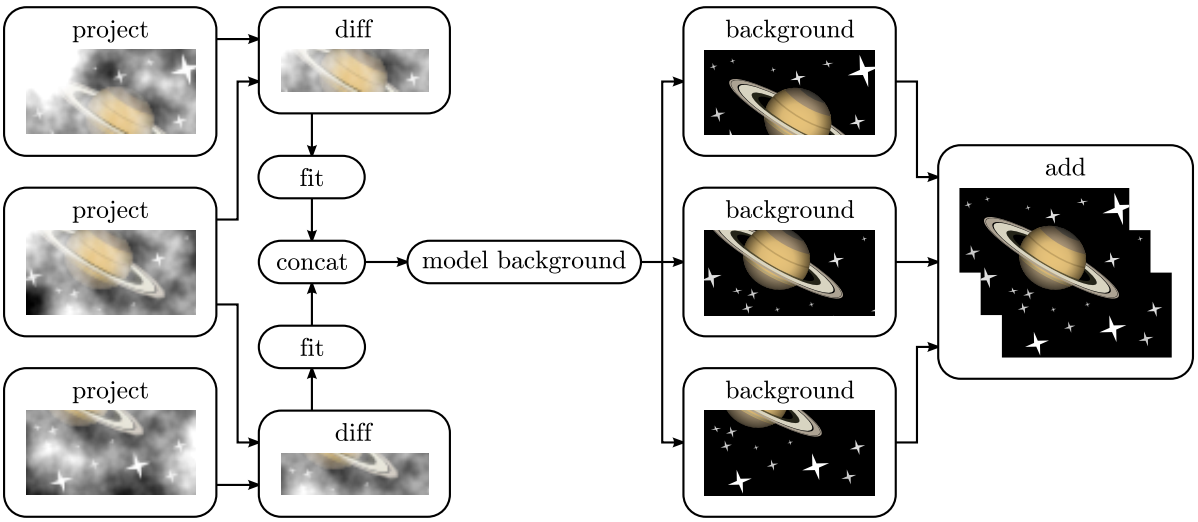


Figure 2.6: An abstract Montage workflow, in which three input images are processed to generate a mosaic. In the workflow, images are first projected to a common scale (project). Overlapping regions in these images are then determined and removed (diff). Subsequently, images are fit on a common plane (fit) and concatenated (concat). Finally, background radiation is corrected (modelBackground, background) and the final mosaic is assembled (add).

generating a Montage workflow, the size as well as the locus of the sky region, for which an image is to be created, can be specified via separate command-line arguments. In addition to generating the workflow file, Montage also downloads any input data required to run this workflows to local storage. See Figure 2.6 for an abstract visualization of workflows generated using the Montage toolkit.

In the first step of a Montage workflow, input images in FITS (Flexible Image Transport System) format – the most commonly used image format in astronomy – are projected to a common spatial scale. To rectify images to a common background level and reduce background noise, background radiation present in the images has to be captured and modeled. To this end, pairs of overlapping images are subtracted from one another. The resulting difference images are then fit on a plane and concatenated before undergoing a modeling of common background radiation. Using this model, background radiation is removed from the input images. Finally, the projected, background-corrected images are merged into the output mosaic in JPEG format.

Montage workflows utilized in evaluations throughout this thesis (see Sections 3.5, 3.6, and 5.2.3) assemble mosaics of the Omega Nebula (i. e., the m17 region of the sky) with varying degree parameters and thus varying volumes of input data.

2.2 Scalable Execution of Scientific Workflows

Scaling scientific workflow execution to increasing amounts of input data necessitates parallelization and distribution of workload. In this section, we outline and contrast

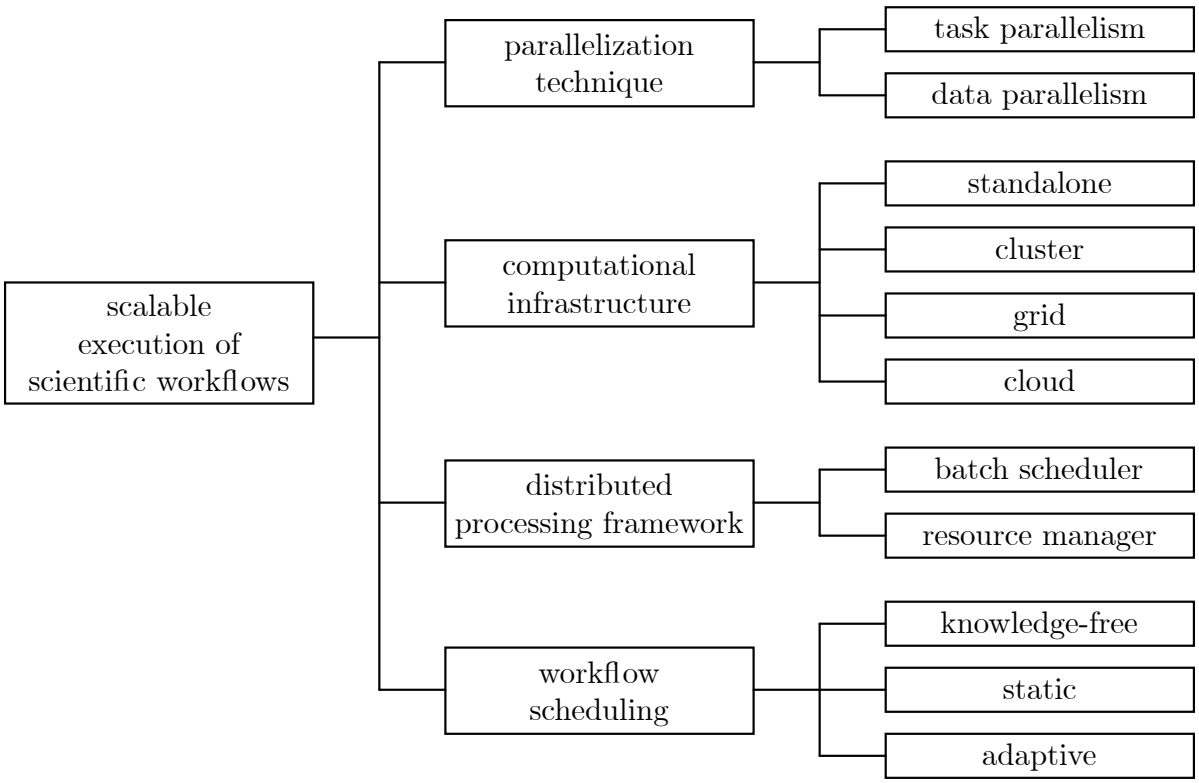


Figure 2.7: A taxonomy on the most fundamental aspects of parallelization and scalable execution of scientific workflows.

fundamental concepts for this purpose, focusing primarily on concepts providing the highest levels of scalability. We differentiate between parallelization techniques, supported computational infrastructures, distributed processing frameworks, and adaptivity in workflow scheduling. We shall adopt these categories to compare different realizations of scalable workflow execution in concrete workflow management systems in Section 2.3. A graphical overview of our taxonomy is given in Figure 2.7.

2.2.1 Parallelization Techniques

Parallelization addresses the question of how to divide the workload incurred by a given scientific workflow for simultaneous computation on multiple processors. A relevant metric in this context is the degree of parallelism.

Definition 4 (Degree of Parallelism) *At any time during the execution of a scientific workflow, the degree of parallelism is defined as the number of tasks that can be run simultaneously, given an unlimited amount of computational resources.*

Since a scientific workflow's tasks and data are treated as black boxes, the degree of parallelism cannot be raised automatically at execution time. Parallelization therefore has to be considered during workflow design.

In general, there are two discernible techniques for the parallelization of data processing: task and data parallelism. In *task parallelism*, processing is partitioned into independent tasks, which can be executed simultaneously. Conversely, in *data parallelism*, data is partitioned into independent fragments, which can be processed simultaneously. These two approaches are not mutually exclusive and are, in practice, often combined with one another.

2.2.1.1 Task Parallelism

By explicitly modeling computation as a graph of tasks (and data), scientific workflows inherently facilitate task-parallel implementations. When executing a given scientific workflow, any number of ready tasks (i. e., tasks whose input data are fully available) can unhesitatingly be executed in parallel. Note that only tasks on parallel branches of the workflow graph can ever have all of their input data available at the same time.

However, in some cases even tasks with data dependencies in between can be executed in parallel by means of pipelining. In *pipelining*, intermediate data is not materialized in the form of files, but is passed directly from task to task, similar to pipelines in Unix-based operating systems or the programming model of stream processing (Gordon et al., 2006). Depending on the scientific workflow as well as the underlying system, pipelining can enable tasks to commence execution without their input data being fully available.

The degree of parallelism that can be achieved through task parallelism is thus limited by (i) how fine-grained computation can be expressed in the form of distinct tasks, (ii) whether pipelining is supported by the workflow's tasks as well as the system executing the workflow, and (iii) whether tasks are arranged sequentially or concurrently.

2.2.1.2 Data Parallelism

The most common approach for achieving data parallelism in scientific workflows is to exploit embarrassingly parallel tasks by implementing the workflow as a bag-of-tasks workflow (see Definition 3 on page 9). An *embarrassingly parallel* task is a task whose input data can be split into fragments to be processed by a bag of tasks, and whose output data can be merged with little to no effort and without influencing the results of computation. Parallelization of such tasks is usually accomplished in one out of two ways, both of which require manually splitting any input data into fragments and manually merging the partial results. The first approach involves feeding the fragments of input data to several explicitly listed tasks of the same bag. The second technique involves specifying the scientific workflows in a data-parallel workflow language such as Cuneiform (Brandt et al., 2015) or Swift (Zhao et al., 2007). The advantage of the latter technique is that tasks belonging to the same bag need not be listed repeatedly and redundantly.

Depending on the granularity of data (i. e., how many parts the data can be feasibly split into), very high degrees of parallelism are achievable. In fact, of the two available techniques for parallelization, data parallelism is the only one able to scale with increasing amounts of input data.

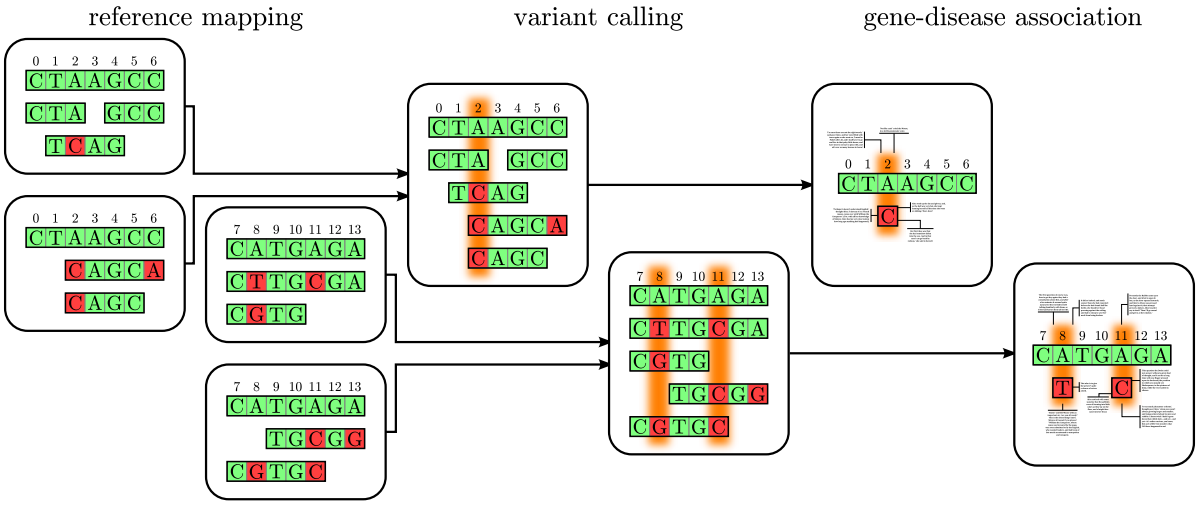


Figure 2.8: An implementation of two-dimensional data parallelism in the workflow introduced in Section 2.1.1.1 and Figure 2.4. All of the workflow's tasks are embarrassingly parallel and can be parallelized by splitting the reference, against which the reads are mapped, into parts. Ideally, these parts correspond to biological entities (e. g., chromosomes). To increase the degree of parallelism further, split sites can also be chosen arbitrarily. In this case, splits have to overlap at their ends, since the split site itself might be part of a valid mapping. While these overlapping ends translate into additional computational effort (due to redundant read mappings), the achievable degree of parallelism is nearly unlimited. As an alternative or in addition to splitting the genomic reference, the input read files can also be split into subsets. However, this latter means of achieving data parallelism is only valid for the reference mapping step, since the subsequent variant calling step requires all mapping data to be present prior to commencing execution.

Note that implementing data parallelism for tasks with multiple incoming (data) edges can lead to substantial amounts of I/O overhead. As an example, consider the abstract SNV calling workflow introduced in Section 2.1.1.1. In a data-parallel implementation of this workflow, which is displayed in Figure 2.8, the reference mapping task expects input in the form of genomic reads and a reference genome. If the reference genome is split into several parts, the complete set of reads has to be parsed and considered for mapping by each of the data-parallel mapping tasks. Depending on whether data is stored locally or over the network, this may translate into a considerable amount of additional disk or network load.

2.2.2 Computational Infrastructures

In this section, we outline the computational infrastructures available for the scalable execution of scientific workflows. Generally, one can differentiate between standalone and distributed architectures. A *standalone architecture* is a system consisting of a

single compute node with its own locally available computational resources in the form of processing cores, volatile primary storage (memory), and persistent secondary storage (hard disks, solid state drives).

Distributed architectures comprise multiple compute nodes connected over a network. While each of these nodes has its own processor, primary or secondary storage are, for some architectures, shared between nodes. Depending on which types of storage (primary, secondary, both, or none) are shared, this type of architecture can be further subdivided into distributed shared-memory, shared-disk, shared-everything, and shared-nothing architectures.

A *distributed shared-nothing architecture* is composed of multiple self-sufficient compute nodes, each with their own processor, primary, and secondary storage. Since this type of architecture does not have a single point of contention and failure, it is able to achieve the highest levels of scalability and shall be discussed in more detail in the remaining parts of this section. For distributed shared-nothing architectures, we differentiate further between computational clusters, grids, and clouds.

Executing a workflow on such clusters, grids, and clouds necessitates (i) supervising the remote execution of a workflow's tasks, (ii) distributing the data read and written by these tasks, (iii) managing (possibly non-uniform) resources fragmented across nodes, and (iv) coping with failures during execution, the likelihood of which increases with the number of compute nodes.

2.2.2.1 Shared-Nothing Cluster Computing

We define a *computational cluster* as a set of tightly coupled, physically co-located compute nodes of usually identical hardware configuration (Bux and Leser, 2013b). Each node has its own operating system, which manages the node's local resources. Nodes are connected to each other over local area networks characterized by low latency and high bandwidth. While such a computational cluster is usually more cost-effective than a standalone system of comparable performance, it entails the increased administrative effort inherent to distributed shared-nothing architectures outlined above.

2.2.2.2 Grid Computing

In scientific research, the need for a large-scale distributed architecture often arises rather sporadically whenever new data is available or new hypotheses ought to be tested. For instance, the single-nucleotide variant workflow outlined in Section 2.1.1.1 processes genomic data, which is typically generated infrequently yet requires substantial computational effort to process. This observation makes investing into costly distributed hardware infeasible for many research groups and has resulted in increased efforts of computational resource sharing by the scientific communities.

Consequently, in the early 1990s, *grid computing* was promoted as a new paradigm of distributed computing in which compute resources of different proprietors were connected in order to solve computationally demanding problems without the need for a supercomputer (Foster et al., 2008). In grid computing (as opposed to cluster com-

puting), compute resources can be heterogeneous and geographically far away from the client. However, similar to the electrical power grid, which the term "grid computing" borrows from, most of this architectural complexity is hidden from the user.

2.2.2.3 Infrastructure-as-a-Service Cloud Computing

Infrastructure-as-a-service *cloud computing* describes a more recently established form of distributed computing (Mell and Grance, 2009). It provides computational resources (e. g., in the form of virtual machines or storage repositories), which can be allocated and accessed over the internet and, in the case of commercial clouds such as Amazon's Elastic Compute Cloud (EC2) or Microsoft Azure, are billed by the minute. Elasticity, which denotes the possibility to adjust the amount of provisioned resources at runtime, constitutes a key benefit of cloud computing. It is particularly promising for the execution of scientific workflows, which, due to their often complex graph structure, can strongly fluctuate in their current computational effort as execution progresses.

Unfortunately, despite commercial cloud vendors providing guarantees with regards to processor clock speed and memory capacity, the actual performance of rented virtual machines varies greatly depending on the configuration of underlying hardware and utilization of shared resources by other users. In Amazon EC2, Dejun et al. (2009) observed response times of CPU- and I/O-intensive web applications to vary by a factor of four and two, respectively. Jackson et al. (2010) found network communication between virtual machines to vary by a factor of up to 1.7 due to sharing of network resources. Zaharia et al. (2008) reported I/O performance to vary by a factor of up to 2.7, depending on how many virtual machines performed I/O operations on the same physical hardware. For scientific workflow management, this finding translates into an elevated importance of adaptivity in scheduling, as outlined in Section 2.2.4. Note that similar (or even more notable) observations of instability can be made for clusters and grids shared between multiple users and organizations (Wolski et al., 2000), though they are much harder to systematically quantify.

Clearly, the execution of scientific workflows presents different challenges depending on the underlying infrastructure. Since resources in computational clusters are tightly coupled, the cost associated with distributing data is less of an issue compared to architectures like grids and clouds. Compute clusters also provide a more homogeneous environment in terms of CPU performance and latency / bandwidth between compute nodes. However, their scalability is more limited and they lack the elasticity provided by computational clouds.

2.2.3 Distributed Processing Frameworks

As outlined in the last section, the resources provided by distributed shared-nothing infrastructures are often heterogeneous and subject to instability. In addition, due to the acquisition and maintenance costs of such infrastructures, they are usually employed for a variety of computational jobs submitted by different users and organizations.

Definition 5 (Job) *A job is a computational workload comprising a collection of tasks, intended for parallel execution by multiple distributed machines.*

Distributed processing frameworks provide the means to manage the execution of such jobs and, in some cases, also the data they process and produce. Essentially, these systems serve as a middleware and layer of abstraction between distributed hardware and heterogeneous jobs. For such distributed processing frameworks, we differentiate between *batch scheduling systems* and *distributed resource managers*.

2.2.3.1 Batch Scheduling Systems

With the advent of grid computing (see Section 2.2.2.2) in the early nineties, a number of systems have been developed for the automated, non-interactive execution of series ("batches") of jobs. Examples for such systems, which are still maintained and employed in practice today, include HTCondor (Litzkow et al., 1988; see also Section 2.3.1.1), Oracle Grid Engine (Gentzsch, 2001), Torque (Staples, 2006), and Slurm (Yoo et al., 2003).

The jobs intended to be dispatched to these systems are typically computationally demanding, yet stable in the amount of resources they consume. Furthermore, the computational infrastructure underlying an installation of these grid-centric systems is often owned by different organizations. For these reasons, scheduling in HTCondor is, for instance, subject to the motto "leave the owner in control, regardless of the cost" (Thain et al., 2005). Consequently, applications running on these systems have only little to no involvement in scheduling decisions.

2.2.3.2 Distributed Resource Managers

In the late 2000s, the introduction of Amazon's commercial cloud service, Elastic Compute Cloud (EC2), as well as the publication and wide-spread adoption of the MapReduce (Dean and Ghemawat, 2008) programming model has altered the landscape of distributed processing frameworks.

A MapReduce program implements a *map* and a *reduce* function, which operate on key-value pairs and can each be executed concurrently for different fragments of input data, i. e., data-parallel (see Section 2.2.1.2). More specifically, each map task reads its block of input data from a distributed file system and applies its map function to the data. Subsequently, each reduce task applies its reduce function to the key-grouped results of the map tasks, storing its output on the distributed file system again. Since both the amount as well as the workload of map and reduce tasks vary and since these tasks are preferably placed on nodes which have their input data available locally, efficiently running MapReduce jobs necessitates a different take on scheduling than the one provided by traditional batch scheduling systems.

These requirements eventuated in the emergence and widespread adoption of a new class of distributed resource management systems, such as Hadoop YARN (White, 2012; Vavilapalli et al., 2013), Mesos (Hindman et al., 2011), and Quasar (Delimitrou and Kozyrakis, 2014). In contrast to batch scheduling systems, these distributed resource

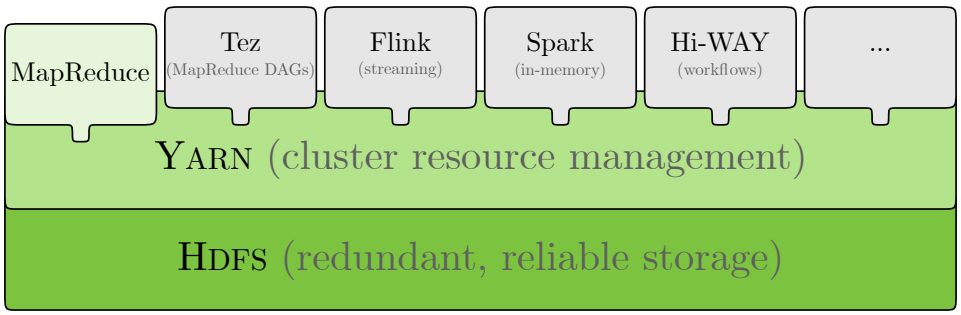


Figure 2.9: The Hadoop software stack, displayed in green colors, comprises HDFS for distributed storage, YARN for resource management, and an application master for running MapReduce jobs. In addition, YARN can be interfaced by a number of external application masters implementing various different programming models.

managers allow for (i) a more fine-grained allocation of resources on the task level as opposed to on the job level (see Definitions 2 and 5 on pages 9 and 19, respectively), (ii) application-controlled scheduling, which enables concepts like data locality and adaptive scheduling (see next section), and (iii) virtualization and containerization of computational resources, which encapsulate resources and give more control to the application (Reuther et al., 2016).

Hadoop In the late 2000s, Hadoop (White, 2012) was developed as an open-source, Java-based implementation of the MapReduce (Dean and Ghemawat, 2008) programming model and distributed file system GFS (Ghemawat et al., 2003). Originally, the management of distributed resources as well as the scheduling and execution of MapReduce jobs were both governed by a singular component called the job tracker, whereas the distributed storage was realized by Hadoop's Distributed File System (HDFS). To increase scalability to levels of tens of thousands of nodes and beyond as well as to enable programming models other than MapReduce to be executed on top of Hadoop, the job tracker was later subdivided into two separate parts. Since version 2.0 and as displayed in Figure 2.9, Hadoop now comprises three built-in components: HDFS, Yet Another Resource Negotiator (YARN), and a MapReduce-specific application master (Vavilapalli et al., 2013).

Chapter 5 of this thesis presents Hi-WAY, an application master for executing scientific workflows on Hadoop. Hi-WAY interfaces with both HDFS and YARN. Concepts relevant for this interaction are therefore introduced here.

HDFS implements a master/slave architecture, in which a single *name node* (NN) manages the file index and supervises data access, interfacing with multiple *data nodes* (DNs), which are slave processes running on the distributed storage nodes. Files stored in HDFS are split into distinct blocks of even size (128 MB by default), such that data-parallel applications like MapReduce jobs can access their to-be-processed data in chunks. These blocks are replicated across multiple data nodes. Three replicas are stored by default,

of which two are placed on the same rack (co-located collection of nodes), whereas the third replica is put on a node of a different rack. This way, data movement is minimized, yet data can be restored in case a node or a whole rack becomes inaccessible.

Similar to HDFS, YARN also employs a master/slave model, in which a single *resource manager* (RM) administers the memory and processing cores provided by multiple *node managers* (NMs) running across the distributed processing nodes. *Containers* are YARN's basic unit of computation, encapsulating a fixed amount of virtual processor cores and memory. Applications running on top of Hadoop can request containers of different configuration from the RM, which, in turn, will determine suitable NMs to host the requested containers. Besides managing distributed resources, YARN introduces the concept of paradigm-specific *application masters* (AMs). An AM for MapReduce jobs is provided by default. Furthermore, Hadoop can be extended with additional AMs.

Every job submitted to Hadoop results in a new AM launched in its own, separate container. This way, scalability is not limited by a single process having to manage all jobs running at the same time, as was the case for the job tracker in Hadoop 1.x. For a visualization of AMs of different programming models interfacing with Hadoop, see Figure 5.3 on page 93.

The flexibility of the system along with its proven scalability has lead to a widespread adoption of Hadoop both in academia and in the industry (see Figure 2.10).

Distributed Dataflow Systems The availability of distributed resource management systems like Hadoop or Mesos has fostered the development of distributed dataflow systems and languages, which emerged from the related field of database research. These systems were designed to efficiently run queries written in SQL-like syntaxes over extremely large data in parallel. In contrast to scientific workflows, they employ white-box (e. g., key-value) data and operator models. These white-box models have the advantage of enabling the automated inference of data parallelism and, potentially, structural graph reordering, as frequently made use of in the field of database query optimization (Rheinländer et al., 2017). However, they come at the cost of reduced flexibility and often require the tools underlying scientific analysis pipelines to be re-implemented. They are therefore ineligible to run arbitrary scientific workflows and will therefore only be briefly outlined here. Besides MapReduce, the following systems and languages have found widespread adoption both in academia and the industry:

- Pig (Olston et al., 2008) provides a workflow-like scripting language that is, at runtime, translated into a series of map and reduce functions.

- Hive (Thusoo et al., 2009) facilitates scalable data aggregation, query, and analysis by means of an SQL-like syntax.

- Spark (Zaharia et al., 2010) and Flink (Alexandrov et al., 2014) support the iterative analysis of in-memory data and the processing of data streams.

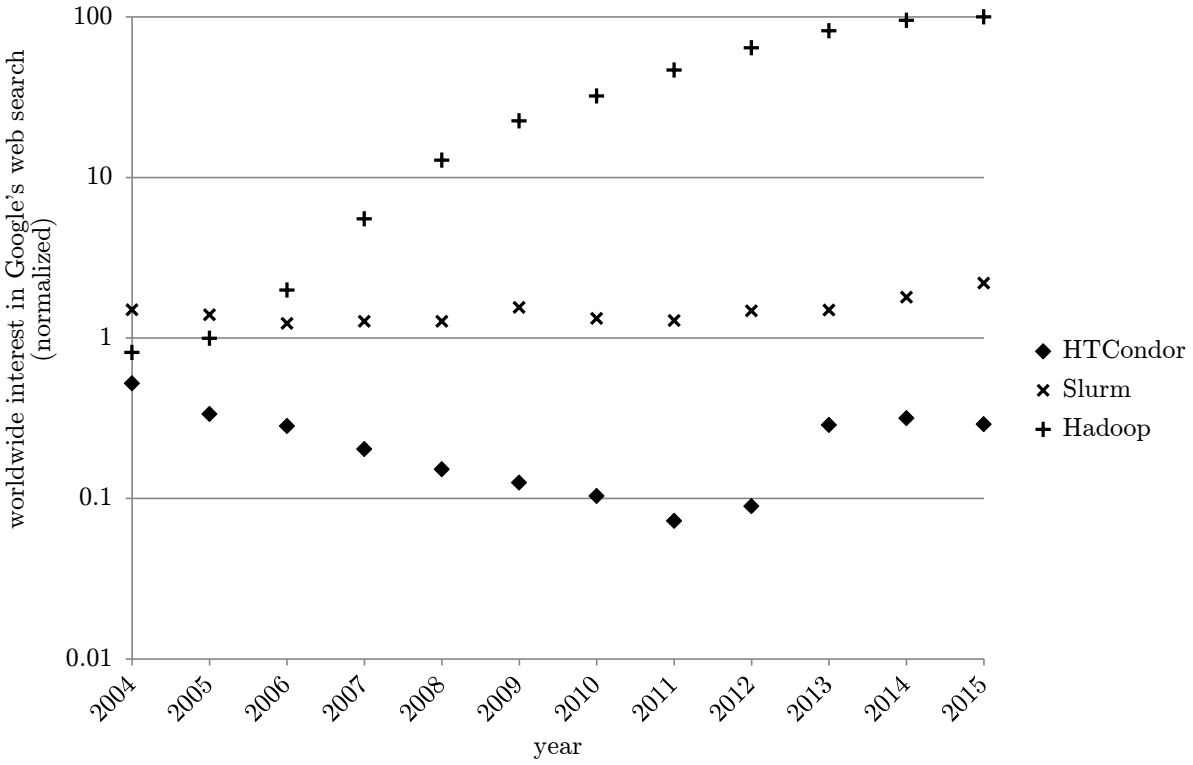


Figure 2.10: Worldwide interest for the topics of HTCondor, Slurm, and Hadoop in Google's web search. Scores are aggregated per year, normalized, and displayed on a logarithmic scale. Notably, the interest in Hadoop has become several orders of magnitude larger than the interest in distributed batch scheduling systems from the last millennium. Numbers provided by Google Trends (www.google.com/trends).

- Tez (Saha et al., 2015) is an extension to MapReduce able to run DAGs of map and reduce tasks without having to stage intermediate data to the distributed file system. It has been primarily developed to accelerate Pig and Hive jobs.

Notably, all of these distributed dataflow systems support execution on top of Hadoop.

2.2.4 Scientific Workflow Scheduling

Scheduling a scientific workflow denotes the process of mapping the workflow's tasks onto the available compute nodes (Mandal et al., 2005). One can generally differentiate between quality-of-service-constrained scheduling and best-effort scheduling (Yu et al., 2008). While the former is driven by completing execution within a given time frame or budget limit, the latter strives to optimize a certain metric or number of metrics without having to meet any hard constraints. The usual goal in best-effort scheduling is to minimize the overall makespan, i. e., wall-clock execution time, of a workflow. However, in certain scenarios it might be preferable to optimize monetary cost investment

or maximize data security. Here, we focus on best-effort scheduling with the goal of minimizing the makespan of a single workflow.

To minimize the makespan of a given workflow, most best-effort schedulers utilize knowledge about the to-be-executed workflow, the underlying computational infrastructure, or both. Usually, this knowledge encompasses estimates for the duration of running any task on any machine. The problem of determining an optimal schedule based on such runtime estimates, formally introduced in Definition 7 on page 59, is NP-complete and is therefore typically approached heuristically (Garey and Johnson, 1979).

As outlined in Section 2.2.2, distributed shared-nothing infrastructures are often heterogeneous and subject to unpredictable performance changes. In such a setting, the performance of a scheduling heuristic (i. e., the reduction in makespan it is able to achieve) largely depends on (i) the currency and accuracy of task runtime estimates as well as (ii) the ability of the heuristic to consider these estimates during scheduling and, therefore, adapt to the computational infrastructure. While the former is addressed in Section 4.4.1, where an overview of methods towards time-series-based task runtime estimation is given, the latter is discussed in this section.

We differentiate between several classes of scientific workflow scheduling: knowledge-free, static, and adaptive scheduling. Here, we shall introduce the mechanisms behind these classes, describe their advantages and disadvantages, and present some typical representatives, which will be referenced at different points in this thesis. A comprehensive overview of available workflow scheduling heuristics has recently been published by Alkhanak et al. (2016).

2.2.4.1 Knowledge-Free Scheduling

A *knowledge-free scheduling* algorithm is oblivious to both the performance characteristics of the computational infrastructure as well as the computational requirements of individual workflow tasks. For this reason, knowledge-free schedulers are easy to implement and often employed in practice. Furthermore, they often serve as a baseline for more elaborate schedulers to evaluate against (Blythe et al., 2005). However, heterogeneity and instability in the underlying computational infrastructure quickly deteriorate the performance of knowledge-free scheduling algorithms.

First-Come-First-Served and Round-Robin The most straightforward approach to knowledge-free scheduling implements a first-come, first-served (FCFS) policy. Here, tasks are placed at the tail of a queue as soon as they're ready to execute. Whenever a resource has an available task slot, it fetches a task from the head of this queue and commences execution.

In round-robin scheduling, another prevalent variant of knowledge-free scheduling, the workflow is traversed from the beginning to the end, assigning tasks to computational resources in turn. This way, each resource will end up with roughly the same amount of tasks, independent of its computational capabilities or the tasks' workload.

2.2.4.2 Static Scheduling

In *static scheduling*, decisions are based on static knowledge such as, especially, task runtime estimates, which are not updated during workflow execution (Yu and Buyya, 2005). Static schedulers assemble a complete, static schedule prior to workflow execution, which is then strictly abided by at runtime. They are able to exploit heterogeneity by determining favorable assignments of tasks to compute nodes. Static scheduling methods can yield good results in controllable and stable compute environments. However, variations in resource performance can strongly impair overall execution time (Rahman et al., 2013).

Min-Min, Max-Min, and Sufferage The Min-Min, Max-Min, and Sufferage heuristics are three polynomial-time, static scheduling heuristics, which have been repeatedly employed in the context of scientific workflow scheduling (Mandal et al., 2005). All of these heuristics iteratively select a task for scheduling that (i) has not been scheduled yet and (ii) has no unscheduled parent tasks. This (greedy) means of static schedule assembly is repeated until all tasks have been scheduled.

In both Min-Min and Max-Min scheduling tasks are selected based on their earliest (minimum) completion time (ECT) across all available machines. Note that a task's completion time on a machine (and, thus, a task's ECT across all machines) depends not only on its (estimated) execution duration on that machine, but also (i) on the ECTs of all of its parent tasks, which have to be executed first and (ii) on the completion time of the last task scheduled on that machine.

Once the ECT values have been determined for all currently unscheduled tasks with no unscheduled parent tasks, the Min-Min (Max-Min) heuristic select the task with the overall minimum (maximum) ECT for scheduling. The selected task is then assigned to the machine on which its completion time was found to be minimal (i. e., on the machine responsible for the determined ECT value). This task-machine-assignment entails possible changes in the ECT values of all remaining unscheduled tasks.

The rationale behind Min-Min scheduling is to keep the overall execution time low by adding workload in minimal increments. On the other hand, the intuition behind Max-Min scheduling is to schedule the most computationally demanding tasks first on their most suitable machines, since smaller tasks contribute less to the workflow's makespan and can likely be executed concurrently with larger tasks.

In Sufferage scheduling, a task's Sufferage value is computed as the difference between its ECT and second-earliest completion time. Hence, a task has a high Sufferage value if it performs really well on one and only one particular machine. Similar to the Max-Min heuristic, the Sufferage heuristic then selects the unscheduled task with the maximum Sufferage value for scheduling. The fundamental idea behind Sufferage scheduling is to favor assigning a machine to a task, which would "suffer" a large degradation in its completion time if another machine were assigned to it.

Heterogeneous Earliest Finishing Time The Heterogeneous Earliest Finishing Time (HEFT) scheduler developed by Topcuoglu et al. (2002) is one of the most established

static scheduling heuristics. HEFT traverses the workflow from the end to the beginning, computing the upward rank of each task as the estimated time to overall workflow completion at the onset of this task. The computation of a given task's upward rank incorporates estimates for both the runtimes and data transfer times of the given task as well as the upward ranks of all successor tasks. The static schedule is then assembled by assigning each task in decreasing order of upward ranks a time slot on a compute node. Hence, HEFT optimizes workflow makespan by mapping tasks with the highest expected time to overall workflow completion onto more suitable (i. e., faster) resources first.

2.2.4.3 Adaptive Scheduling

Performance instability and changes in the availability of computational resources can be problematic if schedules are generated in advance (Rahman et al., 2013). *Adaptive scheduling* denotes the ability to adjust scientific workflow execution to a dynamically changing computational infrastructure at runtime (Casavant and Kuhl, 1988). In contrast to static scheduling heuristics, an adaptive scheduler is therefore able to appropriately cope with instability and volatility and fully unlock the scalability potential of modern distributed architectures, as described in Section 2.2.2. However, this necessitates having available and being able to utilize continuously updated knowledge about the computational infrastructure as well as the tasks composing the scientific workflow. The downsides of adaptive scheduling therefore are that this knowledge (i) can be difficult or computationally intrusive to obtain and (ii) becomes outdated quickly.

Chapter 4 provides an in-depth analysis of the requirements for and benefits to be gained from adaptive scheduling. While we present one particular adaptive scheduling scheme here, Section 4.4.2 provides a comprehensive examination of existing adaptive workflow scheduling heuristics.

Longest Approximate Time to End An example of adaptive scheduling is the LATE (Longest Approximate Time to End) scheduler developed by Zaharia et al. (2008). LATE is a scheduler for MapReduce jobs. It has been developed to provide high levels of robustness to the effects of straggler resources and failed task execution. To this end, LATE keeps track of the elapsed execution times as well as the progress rates of all running tasks. Based on these values, it computes (i) estimates for the remaining execution durations of each task and (ii) estimates for the overall performance of each compute node. LATE then speculatively replicates tasks with the longest approximate time to end on resources performing above a given threshold. By default, LATE reserves 10 % of the computational resources for speculative replication of tasks.

Intuitively, this approach maximizes the likeliness for a speculative copy of a task to overtake its original. LATE exploits heterogeneity in similar fashion as HEFT, since both scheduling heuristics prioritize the assignment of tasks with longest times to finish to well-performing computational resources. LATE has been shown to achieve good results for scheduling MapReduce jobs. However, obtaining reliable progress rates for each task, as required and employed by LATE, is typically not feasible when executing a scientific

workflow, in which tasks are black boxes. Consequently, LATE cannot be employed for scientific workflow scheduling in practice. However, the idea of speculatively replicating tasks during execution is worth exploring for scientific workflows and will be discussed in Chapters 3 and 4 of this thesis.

2.3 Scientific Workflow Management Systems

Due to the complexity of today's typical data analysis pipelines, easy ways of assembling and altering scientific workflows are of major importance (Cohen-Boulakia and Leser, 2011). Moreover, to ensure reproducibility of scientific experiments, workflows should be easily sharable and execution traces should be accessible (Davidson and Freire, 2008). Beginning in the mid 2000s, a number of scientific workflow management systems have been developed to meet these requirements. Examples for established scientific workflow management systems that have undergone continued development include Taverna (Wolstencroft et al., 2013), Pegasus (Deelman et al., 2015), and Galaxy (Goecks et al., 2010).

Definition 6 (Scientific Workflow Management System) *A scientific workflow management system is an application that provides tools for modeling, executing, monitoring, maintaining, and storing scientific workflows.*

While most workflow management systems provide a general-purpose framework for workflow enactment, systems such as Taverna or Galaxy exhibit a certain affinity for a confined field of science and inherently provide domain-specific components. Clearly, there are trade-offs to consider between domain-specific and general-purpose approaches: While confinement to a particular field facilitates use for domain scientists and may help to promote design standards, it necessitates built-in components be kept up to date and limits versatility and interoperability of workflow design.

With scientific workflow management systems reaching maturity, researchers have begun to adopt scientific workflows as a means of specifying and executing complex analysis pipelines. This development is reflected by the growth of public workflow repositories and platforms like myExperiment (Goble and de Roure, 2007). At the same time, ever-increasing quantities of data generated in scientific experiments have elevated the demand for parallel execution of scientific workflows. The growing numbers of cores on servers along with novel computational infrastructures implementing sharing and leasing of compute resources can provide the computational backbone for massively parallel computation.

Adaptive scheduling schemes are necessary to fully leverage the potentials gained from employing distributed infrastructures of large scale. Unfortunately, we shall observe that while there are at least several systems that support execution on distributed computational infrastructures (e. g., Pegasus), only very few systems actually provide a certain degree of adaptivity in workflow scheduling. Altogether, while the scientific workflow community has acknowledged the importance of providing robust systems for highly scalable, possibly heterogeneous infrastructures (Deelman et al., 2017), current systems do not adequately support the necessary functionalities (Liew et al., 2017).

Table 2.1: A categorization of scientific workflow management systems with regard to their supported types of parallelism, distributed processing frameworks (if distribution is supported), and scheduling techniques. Only systems that are actively maintained and used are reported. *Swift supports adaptive load balancing. †Snakemake supports distribution only for shared-storage infrastructures.

system	parallelism	distribution	scheduling	reference
Pegasus	task	batch scheduler	static	Deelman et al. (2015)
Swift	data	batch scheduler	knowledge-free*	Wilde et al. (2011)
Snakemake	data	batch scheduler†	knowledge-free	Köster and Rahmann (2012)
Taverna	data	standalone	knowledge-free	Wolstencroft et al. (2013)
KNIME	data	standalone	knowledge-free	Berthold et al. (2006)
Galaxy	task	batch scheduler	knowledge-free	Goecks et al. (2010)

In this section, we give an overview of support for parallelism and distribution in six established scientific workflow management systems. We illustrate how these systems implement (or fail to implement) the requirements for scalable executions of scientific workflows outlined in Section 2.2, namely whether they (i) facilitate the design of data-parallel workflows, (ii) support any processing frameworks for the distributed execution of workflows, and (iii) employ adaptivity in workflow scheduling. Table 2.1 gives a summary of our findings.

Since we intend to review the current state in scientific workflow management, we focus on open-source systems that were actively maintained at the time of writing and that have been embraced by the scientific community. As a consequence of the latter criterion, we limit the scope of our analysis to systems for which we were able to discover at least a few scientific applications that were published in form of a peer-reviewed article, in which the first author is from a different institution than the workflow system developers (see Table 2.2). Further, due to the vast number of domain-specific solutions, we limit our examination of workflow systems to general-purpose systems (e. g., Pegasus) or systems specific to the life science (e. g., Galaxy). The life sciences serve as a suitable domain to base our analysis on, since many workflow systems have been tailored specifically to the life sciences and a considerable number of computational tasks in the life sciences qualify for parallel execution. For an exhaustive analysis of large-scale scientific workflow management, readers are referred to a recent survey conducted by da Silva et al. (2017).

We distinguish two classes of scientific workflow management systems, which we will outline in separate sections: *textual workflow languages* and *graphical workflow systems*.

2.3.1 Textual Workflow Languages

This category consists of low-level textual languages designed for computer-savvy users adept at using batch scripts and programming languages. Workflows are specified in the form of configuration files, which are interpreted and executed by the workflow man-

Table 2.2: Scientific applications implemented in scientific workflow management systems discussed in this section. Only publications of first authors not involved in the development of the workflow system in question are reported.

system	domain	application	reference
Pegasus	astronomy	mosaics of the sky	Berriman et al. (2004)
	genomics	RNA sequencing	Wang et al. (2011)
	geosciences	climate models	Mayer et al. (2015)
Swift	bioinformatics	protein structure prediction	Adhikari et al. (2012)
	geosciences	climate models	Woitaszek et al. (2011)
Snakemake	life sciences	microscopy data processing	Schmied et al. (2016)
	genomics	RNA sequencing	Wang (2017)
Taverna	genomics	gene regulation analysis	Maleki-Dizaji et al. (2009)
	life sciences	medical image processing	Zhou et al. (2009)
KNIME	chemistry	molecular structure analysis	Saubern et al. (2011)
Galaxy	genomics	RNA sequencing	Wolfien et al. (2016)

agement system. As they were mostly designed to run on heterogeneous, geographically distributed compute resources, considerable administration effort is required for installation and utilization. Hence, the focus of these systems lies less on ease of use and more on efficient computing of heavy workloads.

2.3.1.1 DAGMan and Pegasus

HTCondor (Litzkow et al., 1988; Thain et al., 2005) is a batch job scheduler for high-throughput computing on distributed resources. It puts a strong emphasis on reliability of execution in the form of job checkpointing, recovery, and migration. HTCondor's Directed Acyclic Graph Manager (DAGMan) (Couvares et al., 2007) provides the means to textually specify a workflow as a graph describing a set of tasks along with their data interdependencies. DAGMan then supervises workflow execution, submitting tasks which are ready for execution to HTCondor one at a time. Pipelining or data parallelism are not natively supported since a task is not submitted for execution until all of its input data are available. In case of failure, DAGMan compiles a rescue graph from which execution can be resumed.

DAGMan does not provide the means to automatically set up auxiliary tasks, such as data movement, cleanup, or workflow optimization. To remedy these shortcomings, Deelman et al. (2005, 2015) developed the scientific workflow management system Pegasus on top of DAGMan. Pegasus workflows are specified in a custom XML-based language called DAX, in which any of a workflow's tasks and data are explicitly listed. At runtime, Pegasus transforms a DAX workflow into an executable DAGMan job by adding the aforementioned auxiliary tasks. To this end, Pegasus queries and maintains catalogs of available computational resources, data storage sites, and software libraries. In addition, Pegasus provides capabilities for provenance tracking and execution monitoring, and is able to cluster short-running tasks into joint tasks.

Tasks in the DAGMan input file generated by Pegasus are location-specific, i. e., Pegasus computes a static schedule prior to workflow execution onset. By default, Pegasus provides four different scheduling strategies:

- Random: Tasks are randomly assigned to compute resources able to execute them.
- Round-Robin: Tasks are evenly distributed among resources, independent of the associated computational cost.
- Group: Tasks can be put into user-defined groups. Each group of tasks is scheduled to run on the same compute resource.
- HEFT: Pegasus employs the Heterogeneous Earliest Finishing Time scheduling heuristic described in Section 2.2.4. Runtime estimates for tasks have to be provided by the user.

While Pegasus was originally designed to distribute computationally intensive workflows across grid infrastructures, the growing interest in cloud computing has led to efforts to run Pegasus on cloud infrastructure (Hoffa et al., 2008; Juve et al., 2009; Juve and Deelman, 2011).

In summary, Pegasus supports task-parallel execution of scientific workflows on distributed infrastructures using static scheduling schemes (see Table 2.1). Several computationally intensive workflows from different scientific domains have been implemented and executed in Pegasus (see Table 2.2).

2.3.1.2 Swift

Among the pioneers of parallel workflow execution is the Swift parallel scripting language (Zhao et al., 2007). Swift provides a functional language in which workflows are modeled as a set of program invocations with their associated command-line arguments as well as input and output files. Advanced language constructs such as iteration over lists of data facilitate the design of data-parallel workflows. Swift implements capabilities for failure recovery (retry, restart, and replication) as well as provenance tracking and can be installed both locally and in a distributed setting.

Scheduling in Swift is knowledge-free, i. e., Swift does not differentiate between different tasks when dispatching them for execution. However, Swift does employ an adaptive load balancing scheme, which, via trial and error, determines how many tasks can safely be assigned to a given resource: For each known compute resource, the Swift execution engine maintains a score which increases with each successful task execution and decreases with each failure due to overload (Wilde et al., 2011). Tasks are assigned to compute resources at runtime and the higher the score of a resource the more tasks will be assigned to it.

In summary, Swift implements data-parallel workflow execution on distributed architectures, as shown in Table 2.1. It has been utilized for computationally intensive applications from various fields of science, including physics and the life sciences (see Table 2.2).

2.3.1.3 Snakemake

Köster and Rahmann (2012) developed Snakemake as a light-weight and flexible text-based workflow management system for bioinformatics workflows. It provides its own workflow specification language, which is inspired by Python as well as GNU make, enabling a goal-driven assembly of workflow scripts. Tasks within a Snakemake workflow contain either native Python code or arbitrary shell commands.

Snakemake supports regular expressions for specifying valid names for the files processed and produced by a given task. These regular expressions are then employed to automatically determine data dependencies between tasks, i. e., for matching a task's output files to another task's input files. This functionality enables the flexible and lean design of data-parallel scientific workflows.

Snakemake does not provide its own distributed execution environment. However, it can interface with distributed batch schedulers that are able to execute shell scripts and have access to a common file system. This encompasses batch schedulers running on top of a shared-storage or shared-everything cluster, as defined in Section 2.2.2. In doing so, Snakemake leaves task placement decisions to the underlying distributed processing engine, providing no means of distributed scheduling by itself.

As shown in Table 2.1, Snakemake supports the execution of data-parallel workflows on distributed shared-storage infrastructures. To this end, it is able to interface with batch schedulers, to which it leaves any scheduling decisions.

2.3.2 Graphical Workflow Systems

The class of graphical scientific workflow management systems comprises systems with a strong emphasis on ease of use and graphical representation of workflows. They provide a graphical user interface for workflow design and execution monitoring as well as a range of general-purpose and often domain-specific task libraries. Some systems (e. g., Galaxy) even provide a web portal and public servers, where scientists can design, execute, and share workflows. Since graphical representation becomes problematic for large workflows consisting of hundreds or thousands of tasks (Deelman et al., 2009), most graphical systems support the hierarchical nesting of subworkflows. Graphical systems sometimes support multithreading, but most of them are not able to utilize external compute resources by default, save via invoking tasks that interface with resources outside of the workflow management system's control (e. g., web services).

2.3.2.1 Taverna

Taverna workbench is a graphical workflow management system primarily developed for the enactment of bioinformatics workflows comprising (usually short-running) local tasks and web service invocations (Oinn et al., 2004; Missier et al., 2010; Wolstencroft et al., 2013). It focuses on usability, providing a graphical user interface for workflow design and monitoring as well as a comprehensive collection of pre-defined tools and

remote services. Recently, Taverna workbench has been superseded by Apache Taverna, a command-line-based workflow system based on the same internals.

Taverna emphasizes reproducibility of experiments and workflow sharing by integrating the public myExperiment workflow repository (Goble and de Roure, 2007), in which over a thousand workflows have been made available. Taverna can utilize only a single (local) machine or a remote server; computation on distributed architectures is not supported. Scheduling in Taverna is approached greedily by means of a FCFS policy.

Taverna parts from the black-box data model inherent to most scientific workflow systems by operating on structured data in the form of lists. If a task receives a list of data items on an input port where a single item is expected, each element of the list is processed by a replicate of the task in a separate thread (Missier et al., 2010). Each processed data item is passed to follow-up tasks for immediate consumption in a new thread. Taverna's execution model therefore not only supports task parallelism in the form of pipelining, but even implements data parallelism, since it allows multiple replicate data processing pipelines to run concurrently.

In summary, while Taverna supports data-parallel execution of workflows, it is devoid of sophisticated scheduling techniques and can currently only utilize cores on a single local resource (see Table 2.1). Several computationally intensive problems from the field of bioinformatics have been implemented in Taverna, as shown in Table 2.2.

2.3.2.2 KNIME

The Konstanz Information Miner (KNIME) shares many characteristics with Taverna, albeit with a stronger focus on user interaction and visualization of results and less emphasis on web service invocation (Berthold et al., 2006; Sieb et al., 2007). Furthermore, in contrast to all other systems described here, KNIME provides commercial licenses and proprietary extensions for business customers. KNIME focuses on workflows from the fields of data mining, machine learning, and chemistry, for which it provides a range of libraries and pre-built components. A graphical user interface facilitates design and execution monitoring of workflows. KNIME can either be installed locally or on a standalone server, accessible via multiple clients.

To achieve data parallelism, KNIME requires the designer of a task node to explicitly specify whether it qualifies for data parallel execution. If implemented accordingly, KNIME automatically splits the entire input data into four times as many chunks as the size of the thread pool, which is restricted via user-defined constraints. Each chunk is then processed by a replicate task and aggregate results are merged as soon as all threads have finished execution (Sieb et al., 2007). Similar to Taverna, scheduling is conducted knowledge-free by means of a FCFS policy.

As shown in Table 2.1, KNIME implements knowledge-free scheduling as well as data parallelism in the form of multiple threads on a local machine or on a remote server. Distributed architectures like a compute grid or cloud are not supported directly. However, connectors for invoking external Spark jobs or Hive queries, potentially running on a distributed infrastructure, have recently been open-sourced.

2.3.2.3 Galaxy

With the advent of next-generation sequencing and the general growth of data in the life sciences, Galaxy (Goecks et al., 2010) has been established as one of the major frameworks for genomic research in the life sciences. Galaxy comes with a web-based graphical user interface as well as assorted pre-built components for common tasks in sequence analysis. Workflows can be assembled from task and data repositories, shared with other users and executed on a public or on a private server. See Figure 5.9 on page 104 for a Galaxy workflow developed by Wolfien et al. (2016).

Afgan et al. (2010) developed CloudMan as an extension to Galaxy, which enables the deployment of Galaxy installations on distributed cloud resources. While CloudMan instances can be set up on Amazon EC2 through an easy-to-use web interface, scalability is limited to a maximum of 20 compute nodes. CloudMan can also be employed for cloud bursting, i. e., a provisioning of additional cloud resources on-demand when the workload of a local Galaxy installation exceeds local resource capabilities (Afgan et al., 2015).

Galaxy implements a white-box data model, which facilitates workflow design and visualization of results. However, despite the prevalence of embarrassingly parallel problems in computational biology, for many of which Galaxy provides pre-built tools, Galaxy is not able to automatically infer potentials for data-parallel workflow execution. See Table 2.1 for on overview of Galaxy's capabilities regarding scalable workflow execution.

2.4 Bridging the Scalability Gap

As outlined in Section 2.2, scaling the execution of scientific workflows to ever-growing volumes of input data necessitates the utilization of data parallelism and distributed shared-nothing infrastructures. Distributed resource management and adaptive workflow scheduling are crucial to cope with the instability and exploit the heterogeneity typically encountered on such infrastructures. Unfortunately, as observed in Section 2.3, established scientific workflow management systems provide only very limited support for these concepts. Consequently, we observe a widening gap between the prevalence of data-intensive scientific workflows and the systems available for their execution.

In the exemplary case of bioinformatics, this gap finds expression in (i) increasingly voiced demands for bioinformaticians to discontinue using traditional programming languages and shift to programming models and languages implementing a white-box data model instead (Nekrutenko and Taylor, 2012; Prins et al., 2015), and (ii) the emergence of various makeshift solutions, which often trade the flexibility of the scientific workflow programming model for increased levels of scalability, e. g., by only supporting execution of a limited number of pre-built workflows.

The remaining parts of this section give an overview of these two commonly proposed approaches. Conversely, closing the aforementioned gap without compromising the scientific workflow programming model serves as the key motivation for this thesis and solutions will be discussed in subsequent chapters.

2.4.1 Re-Implementations in White-Box Data Systems

Several computationally intensive analysis methods in bioinformatics have recently been implemented using the MapReduce programming model. Examples include MapReduce-based re-implementations of individual sequence mapping and variant calling tools, such as CloudBurst (Schatz, 2009), BigBWA (Abuín et al., 2015), Seal (Pireddu et al., 2011), or Crossbow (Langmead et al., 2009a). Also, implementations of whole workflows as sequences of MapReduce jobs have been proposed, such as Halvade (Decap et al., 2015). A survey of bioinformatics toolkits using the MapReduce programming model has been conducted by Zou et al. (2013).

A number of software libraries have been developed as extensions of the distributed dataflow languages presented in Section 2.2.3. For instance, SeqPig (Schumacher et al., 2014) and BioPig (Nordberg et al., 2013) provide re-implementations of bioinformatics tools as an extension of Pig. Similarly, Adam (Massie et al., 2013; Nothaft et al., 2015) and SparkSeq (Wiewiórka et al., 2014) build on top of Spark.

Since installing and utilizing MapReduce-based applications on a distributed computational infrastructure might be difficult for domain scientists, Schoenherr et al. (2012) developed Cloudgene as an extensible execution environment for MapReduce programs in bioinformatics. Cloudgene features a graphical user interface and a selection of built-in components. It allows the graphical design and distributed execution of analysis pipelines on local clusters or public clouds, such as Amazon EC2.

2.4.2 Workarounds and Makeshift Solutions

The scientific workflow middleware SciCumulus allows the outsourcing of the most computationally demanding tasks of a workflow to external resources (de Oliveira et al., 2010). To this end, scientists can wrap computationally intensive tasks of their existing workflows into SciCumulus cloud activities. SciCumulus provides components for upload, dispatch, download, and provenance capture. These components can be interfaced from within a workflow management system like Taverna or Pegasus. By using predefined or custom cartridges, users can specify how data be fragmented before and merged after processing for data-parallel computation. Notably, SciCumulus also provides an adaptive scheduling mechanism that evaluates the performance of compute nodes in a possibly heterogeneous setting by (i) running small evaluation routines and (ii) slowly increasing the input data of workflow tasks, observing for performance degradation (de Oliveira et al., 2012).

In the light of overwhelming assortments of bioinformatics applications and libraries, users might prefer to employ pre-built workflows instead of assembling their own data processing pipelines. Following this rationale, Angiuoli et al. (2011) developed the Cloud Virtual Resource (CloVR), a life science gateway featuring a selection of several hard-coded workflows covering some of the major tasks in next-generation sequence analysis. These workflows are encased by a virtual machine image and are therefore easy to set up and execute. Computationally intensive reference mapping steps occurring in three of the workflows are split and distributed among dynamically extendable cloud resources.

2.5 Summary

Scientific workflows provide a flexible means of processing scientific data. However, a growing interest of the scientific community in data-driven research has eventuated in increasingly high requirements of computational power and an elevated need for workload distribution.

In this chapter, we introduced the concept of bag-of-tasks workflows and presented three well-established data-intensive bag-of-tasks workflows from the fields of computational genomics and astronomy. We then discussed several design choices for parallelizing scientific workflows, i. e., parallelization techniques, computational infrastructures, distributed processing frameworks, and (adaptivity in) workflow scheduling. Subsequently, we outlined the shortcomings of established scientific workflow management systems with regard to these concepts.

We argue that scientific workflow management systems have to adapt to stay competitive: (i) data-parallel workflow design must be facilitated, (ii) highly scalable distributed infrastructures, such as infrastructure-as-a-service clouds, should be natively supported, (iii) well-established distributed resource managers like Hadoop YARN have to be supported, and (iv) adaptive scheduling policies have to be developed and employed in practice.

As a first step in the development of novel adaptive scheduling policies, a simulation framework would be valuable, since simulation is an often employed first step in evaluating resource provisioning and scheduling algorithms. In the next chapter, we therefore present DynamicCloudSim, a simulation toolkit that mimics the instability and performance variability inherent to computational clouds and similar shared distributed infrastructures.

3 Simulating Instability in Computational Clouds

Over the last decade, cloud computing emerged as a form of distributed computing, in which computational resources are provisioned on-demand over the Internet (see Section 2.2.2.3). In the commercial infrastructure-as-a-service model of cloud computing, computational resources in the form of virtual machines of any scale can be rented on demand from cloud providers like Amazon or Microsoft (Foster et al., 2008). The convenience of its pay-as-you-go billing model, in which computational resources are paid by the minute, has led to a substantial growth in the usage of cloud computing over the last years (see Figure 3.1).

Tailoring resource-demanding applications to make efficient use of cloud resources requires developers to be aware of both the performance of the used cloud infrastructure as well as the requirements of the to-be-deployed application. These characteristics are hard to quantify and vary depending on the application and cloud provider. Benchmarking a given application on any given infrastructure of large scale repeatedly under various configurations is both tedious and expensive. Simulation therefore constitutes a convenient and affordable way of evaluation prior to deployment on real hardware (Beloglazov and Buyya, 2012; Wu et al., 2011; Sadhasivam et al., 2009). It has also been repeatedly made use of for evaluating scientific workflow schedulers (Braun et al., 2001; Blythe et al., 2005).

Unfortunately, available cloud simulation toolkits like CloudSim (Calheiros et al., 2011) or GroudSim (Ostermann et al., 2010) do not adequately capture inhomogeneity and dynamic performance changes inherent to non-uniform and shared infrastructures like computational clouds. The effect of these factors of variability and instability is not negligible and has been repeatedly observed to strongly influence the runtime of applications (Zaharia et al., 2008; Ostermann et al., 2008; Palankar et al., 2008; Dejun et al., 2009; Jackson et al., 2010; Schad et al., 2010; Iosup et al., 2011; Novaković et al., 2013; Maji et al., 2014; Lloyd et al., 2017).

In this chapter, we present DynamicCloudSim[1], an extension to CloudSim which models the variability and instability inherent to computational clouds and other distributed shared-nothing infrastructures. Section 3.1 gives an overview of the different aspects of performance variability encountered in such infrastructures as a consequence of heterogeneous hardware and virtual machine interference. These aspects of variability comprise (i) heterogeneity between virtual machines of similar configurations, (ii) dynamic changes of performance at runtime, and (iii) straggler machines and failures during execution.

[1]The code of DynamicCloudSim is available at `https://github.com/marcbux/dynamiccloudsim`

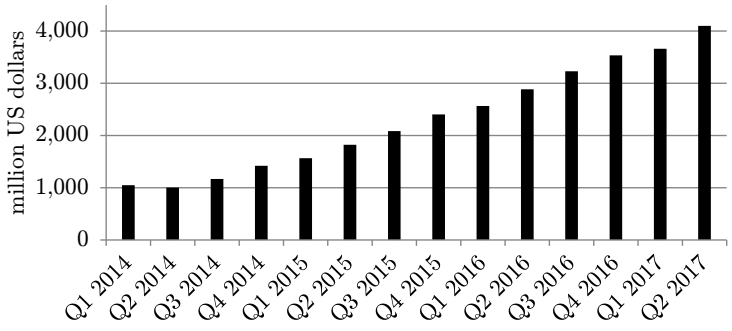


Figure 3.1: Reported quarterly revenue of Amazon's commercial cloud Amazon Web Services (AWS).

We give a description of the architecture and feature set of the CloudSim framework in Section 3.2. Subsequently, Section 3.3 outlines the extensions provided by DynamicCloudSim, which include models for all of the aforementioned aspects of variability. We describe how to simulate the execution of scientific workflows on a datacenter resembling Amazon EC2 in Section 3.4. In Section 3.5, we evaluate the applicability of DynamicCloudSim in a series of experiments involving the execution of two computationally intensive scientific workflows both in simulation and on real cloud infrastructure.

A number of algorithms for scheduling scientific workflows on distributed infrastructures have been developed (see Section 2.2.4). As an application example for DynamicCloudSim, we compare the performance of several established scientific workflow schedulers at different levels of instability in Section 3.6. Since some of the investigated schedulers have been developed to handle resource heterogeneity, dynamic performance changes, and failure, we expect our experiments to replicate the advertised strengths of the different workflow schedulers. Results from an extensive number of simulation runs confirm these expectations, underlining the importance of elaborate scheduling mechanisms when executing workflows on distributed infrastructure which are subject to resource contention and performance variation. Related work to DynamicCloudSim is discussed in Section 3.7 and a summary of DynamicCloudSim is presented in Section 3.8.

3.1 Performance Variations in Commercial Clouds

In a performance analysis spanning multiple Amazon EC2 datacenters, Dejun et al. (2009) observed occasional severe performance drops in virtual machines, which would greatly increase the response time of running tasks. Furthermore, they reported the response time of CPU- and I/O-intensive application to vary by a factor of up to four on virtual machines of equal configuration. Notably, they detected no significant correlation between CPU and I/O performance of virtual machines. Zaharia et al. (2008) found the I/O throughput of "small"-sized virtual machine instances in EC2 to vary between roughly 25 and 60 MB per second, depending on the amount of co-located virtual machines running I/O-heavy tasks.

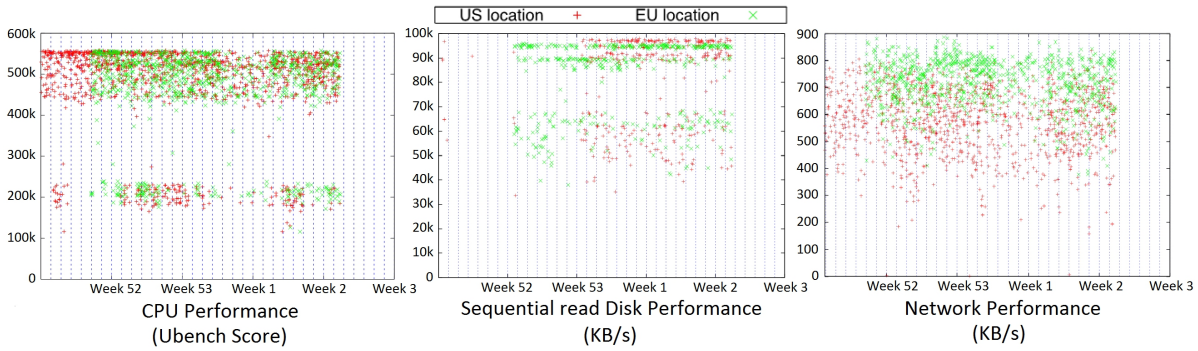


Figure 3.2: CPU, sequential read disk, and network performance of Amazon EC2 large instances, measured over the course of several weeks. Each dot represents the performance of a single virtual machine at a certain point in time. Images taken from (Schad et al., 2010) with friendly permission.

In a similar evaluation of Amazon EC2, Jackson et al. (2010) detected different physical CPUs underlying similar virtual machines: Intel Xeon E5430 2.66 GHz, AMD Opteron 270 2 GHz, and AMD Opteron 2218 HE 2.6 GHz. They also observed network bandwidth and latency to depend on the physical hardware of the provisioned virtual machines. When executing a communication-intensive task on 50 virtual machines, the overall communication time varied between three and five hours over seven runs, depending on the (unknown) network architecture underlying the provisioned virtual machines. In the course of their experiments, they also had to restart about one out of ten runs due to failures. Similar observations have been made in other studies on the performance of cloud infrastructure (Ostermann et al., 2008; Palankar et al., 2008).

Another comprehensive analysis of the performance variability in Amazon EC2 was conducted by Schad et al. (2010). Once per hour over a time period of two months they benchmarked the CPU, I/O, and network performance of newly provisioned virtual machines in Amazon EC2. Performance was found to vary considerably and generally fall into two bands, depending on whether the virtual machine would run on Intel Xeon or AMD Opteron infrastructure (see Figure 3.2). The variance in performance of individual virtual machines was also shown to strongly influence the runtime of a real-world MapReduce application on a virtual cluster consisting of 50 EC2 virtual machines. A further interesting observation of this study was that the performance of a virtual machine depends on the hour of the day and day of the week. Iosup et al. (2011) made similar observations when analyzing more than 250,000 real-world performance traces of commercial clouds.

Evidently, the performance of computational cloud infrastructure is subject to different aspects of performance variability:

1. Heterogeneous physical hardware underlying provisioned virtual machines (HET)
2. Dynamic changes of performance at runtime (DCR)
3. Straggler virtual machines and failed task executions (SAF)

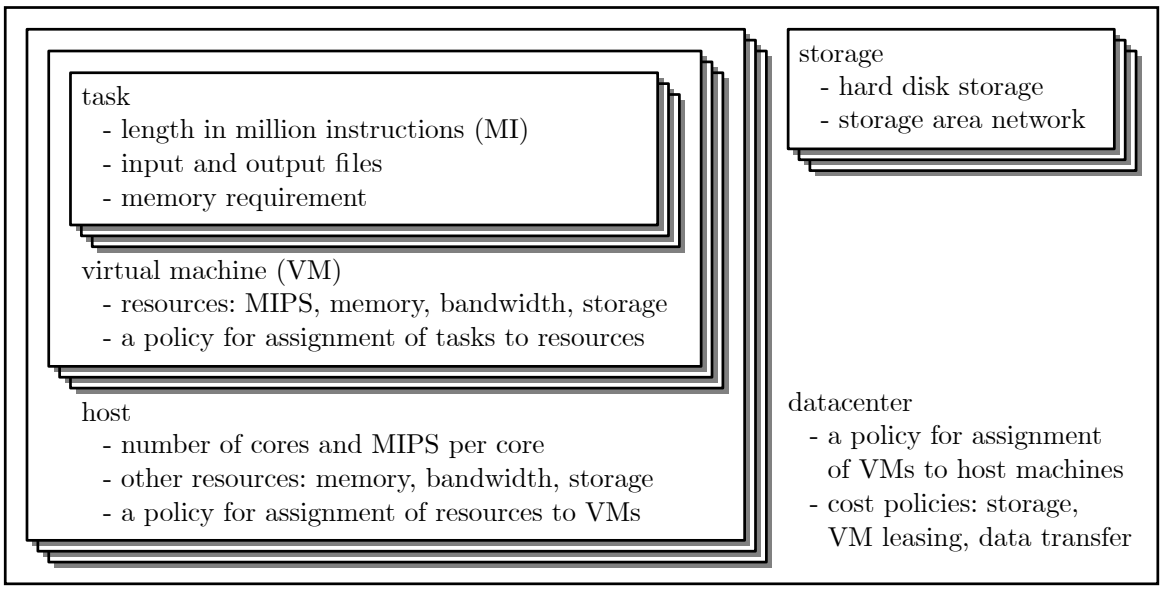


Figure 3.3: The architecture of the CloudSim framework. A datacenter comprises several storage nodes along with hosts providing computational resources. Hosts can spawn virtual machines, which can be assigned tasks for execution.

3.2 CloudSim

CloudSim is an extension of the GridSim (Buyya and Murshed, 2002) framework for simulation of resource provisioning and scheduling algorithms on cloud computing infrastructure. It was developed by Calheiros et al. (2011) at the University of Melbourne's CLOUDS Laboratory. It provides capabilities to perform simulations of assigning and executing a given workload on a stable cloud computing infrastructure under different experimental conditions. CloudSim, for instance, has been used (i) to measure the effects of a power-aware virtual machine provisioning and migration algorithm on datacenter operating costs for real-time cloud applications (Beloglazov and Buyya, 2012), (ii) to evaluate a cost-minimizing algorithm of virtual machine allocation for cloud service providers that takes into account a fluctuating user base and heterogeneity of cloud virtual machines (Wu et al., 2011), and (iii) to develop and showcase a scheduling mechanism for assigning tasks of different categories – yet without data dependencies – to the available virtual machines (Sadhasivam et al., 2009).

CloudSim operates event-based, i. e., all components of the simulation maintain a message queue and generate messages, which they pass along to other entities. A CloudSim simulation can instantiate several *datacenters*, each of which comprises *storage* servers and physical *host* machines, which in turn host multiple *virtual machines* executing several *tasks* (named *cloudlets* in CloudSim). For a detailed overview, refer to Figure 3.3.

A datacenter is characterized by its policy of assigning requested virtual machines to host machines (the default strategy being to always choose the host with the least cores in use). Each datacenter can be configured to charge different costs for storage, virtual machine usage, and data transfer.

The computational requirements and capabilities of hosts, virtual machines, and tasks are captured in four performance measures: MIPS (million instructions per second) per core, bandwidth, memory, and local file storage. Furthermore, each host has its own policy which defines how its computational resources are distributed among allocated virtual machines, i. e., whether virtual machines operate on shared or distinctly separated resources and whether over-subscription of resources is allowed. Similarly, each virtual machine comes with a scheduling policy specifying how its resources are distributed between tasks. On top of this architecture, an application-specific datacenter broker supervises the simulation, requesting the (de-)allocation of virtual machines from the datacenter and assigning tasks to virtual machines.

One of the key aspects of CloudSim is that it is easily extensible. Several extensions have been presented, including (i) NetworkCloudSim (Garg and Buyya, 2011), which introduces sophisticated network modeling and inter-task communication, (ii) EMUSIM (Calheiros et al., 2013), which uses emulation to determine the performance requirements and runtime characteristics of an application and feeds this information to CloudSim for more accurate simulation, or (iii) CloudMIG (Frey and Hasselbring, 2011), which facilitates the migration of software systems to the cloud by contrasting different cloud deployment options based on simulations in CloudSim.

3.3 DynamicCloudSim

CloudSim assumes provisioned virtual machines to be predictable and stable in their performance: Hosts and virtual machines are configured with a fixed amount of MIPS and bandwidth and virtual machines are assigned to the host with the most available MIPS. On actual cloud infrastructure like Amazon EC2, these assumptions do not hold (see Section 3.1).

Most infrastructure-as-a-service cloud vendors guarantee a certain processor clock speed, memory capacity, and local storage for each provisioned virtual machine. However, the actual performance of a given virtual machine is subject to the underlying physical hardware as well as the usage of shared resources by other virtual machines assigned to the same host machine. In this section, we outline the extensions we have made to the CloudSim core framework as well as the rationale behind them.

3.3.1 Fine-Grained Resource Modeling

In CloudSim, the amount of time required to execute a given task on a virtual machine depends solely on the task's length (in MI) and the virtual machine's processing power (in MIPS). Additionally, the network throughput (in KB/S) of virtual machines and of their host machines can be specified, but neither have an impact on the runtime of a task. However, many data-intensive tasks are not CPU-bound, but primarily I/O- or network-bound. Especially in database applications, a substantial amount of tasks involves reading or writing large amounts of data to local or network storage (Dejun et al., 2009).

DynamicCloudSim therefore provides a more fine-grained representation of computational resources to allow for the simulation of executing I/O- or network-bound tasks. To this end, it implements network and disk data transfer (both in KB) as additional resource requirements of tasks besides CPU operations (in MI). Furthermore, it introduces disk I/O throughput (in KB/s) as a further resource provided by virtual machines and hosts besides CPU performance (in MIPS) and network throughput (in KB/s). Consequently, in contrast to CloudSim, DynamicCloudSim enables simulation experiments, in which virtual machines provide and tasks require varying amounts of CPU, disk, and network resources. During a simulation run, DynamicCloudSim takes into account all performance requirements of a task when determining how long it takes to execute the task on a given virtual machine.

3.3.2 Heterogeneity

Similar to Amazon EC2, the provisioning of computational resources to virtual machines in DynamicCloudSim is based on the abstract notion of *compute units* instead of concrete performance metrics such as GHz or MIPS. A virtual machine is characterized by a number of compute units and an amount of memory. Conversely, a host provides a number of compute units, an amount of processing power per compute unit (in MIPS), an amount of memory (in MB), and a total I/O and network throughput (in KB/s). DynamicCloudSim assigns new virtual machines to random hosts within the datacenter that are able to provide the requested amount of compute units and memory. A virtual machine running on a host therefore has its own allotted number of compute units and, thus, MIPS. However, it shares the host's I/O and network throughput with other virtual machines. Furthermore, two virtual machines with the same number of designated compute units might be provided with different amounts of actual MIPS depending on what host machine they are placed on.

In addition to this heterogeneity (HET) introduced through the concept of compute units, DynamicCloudSim also provides the functionality to randomize the performance characteristics of a virtual machine (i. e., its share of the host's CPU, I/O, and network resources). Schad et al. (2010) found several of the performance measurements of virtual machines in Amazon EC2 – particularly random disk I/O throughput and network bandwidth – to be normally distributed. DynamicCloudSim therefore allows for the performance characteristics of a virtual machine to be sampled from a normal distribution instead of using the default values defined by the virtual machine's host machine. The mean of this normal distribution is set to the host's default value of the respective performance characteristic and the relative standard deviation (RSD) can be defined by the user, depending on the desired level of heterogeneity.

Dejun et al. (2009) reported average runtimes between 200 and 900 ms – with a mean of roughly 500 and a standard deviation of about 200 – for executing a CPU-intensive task on 30 virtual machines across six Amazon EC2 datacenters. Based on these measurements, we set the default value for the RSD parameter responsible for CPU performance heterogeneity to 0.4. In the same way, we determined a default value for I/O heterogeneity of 0.15 based on findings by Dejun et al. (2009). Based on measurements

Table 3.1: An excerpt of parameters provided by DynamicCloudSim for configuring a simulation experiment. *Parameter is configurable separately for each type of resource, i. e., for CPU performance, I/O throughput, and network throughput. †Available distributions are exponential, gamma, log-normal, Lomax, normal, Pareto, uniform, Weibull, and Zipf. ‡Parameters depend on the selected distribution, e. g., relative standard deviation (RSD) for normal distribution.

parameter group	parameter	default value(s)
HET parameters	performance baseline distribution*†	normal
	performance baseline RSD*‡	0.4, 0.15, 0.2
DCR parameters	average persistent performance changes per hour*	0.5
	persistent performance changes distribution*†	normal
	persistent performance changes RSD*‡	0.054, 0.033, 0.04
	short-term performance fluctuations distribution*†	normal
	short-term performance fluctuations RSD*‡	0.028, 0.007, 0.01
SAF parameters	likelihood of straggler VM	0.015
	performance factor for straggler VM	0.5
	likelihood of failed task execution	0.002
	task runtime factor in case of failure	20

taken by Jackson et al. (2010) for communication-intensive tasks on Amazon EC2, we set the default value for network bandwidth heterogeneity to 0.2. These values are backed up by the performance measurements of Schad et al. (2010), who observed an RSD of 0.35 between processor types in EC2 as well as RSD values of 0.2 and 0.19 for disk I/O and network performance.

See Table 3.1 for an overview of the configurable parameters in DynamicCloudSim.

3.3.3 Dynamic Changes at Runtime

So far we have only modeled heterogeneity between virtual machines, which represents permanent variability in performance of virtual machines due to differences in underlying hardware. Another important property of cloud infrastructures are dynamic changes of performance characteristics at runtime (DCR) as a consequence of virtual machine colocation and interference (Dejun et al., 2009). DynamicCloudSim models such dynamic changes in two ways: (i) persistent (long-term) changes in a virtual machine's performance due to a certain event, e. g., the co-allocation of a different virtual machine with high resource utilization on the same host and (ii) noise and short-term fluctuations in a virtual machine's performance. DynamicCloudSim does not explicitly model external factors affecting virtual machine performance, but implicitly models their effects on performance.

To simulate the effects of long-term changes, DynamicCloudSim samples from an exponential distribution with a given rate parameter to determine the time of the next performance change. The exponential distribution is frequently used to model the time

between state changes in continuous processes. In DynamicCloudSim, the rate parameter is defined by the user and corresponds to the average number of performance changes per hour. In light of the observations made by Schad et al. (2010) and Iosup et al. (2011), who found that the performance of a virtual machine can change on an hourly basis (but does not necessarily do so), we assume the performance of a virtual machine to change about once every other hour by default. Since this parameter is highly dependent on the particular computational infrastructure, we encourage users to adjust it to their respective environment.

Whenever a change of a performance characteristics has been induced on a virtual machine, the new value for the given characteristic is, by default, sampled from a normal distribution, though DynamicCloudSim also supports the use of other distributions. The mean of this normal distribution is set to the baseline value of the given characteristic for this virtual machine, i. e., the value that has been assigned to the virtual machine at allocation time. The RSD of the distribution can be adjusted through a parameter. Higher values in both the rate parameter of the exponential distribution and the standard deviation of the normal distribution correspond to higher levels of dynamics.

Noise and short-term fluctuations are another, albeit less impactful, source of dynamic performance changes in DynamicCloudSim. These fluctuations are modeled by introducing slight aberrations to a virtual machine's performance whenever a task is assigned to it. As with heterogeneity and long-term performance changes, this is by default achieved by sampling from a normal distribution with user-defined RSD parameter.

On Amazon EC2, Dejun et al. (2009) observed relative standard deviations in performance between 0.019 and 0.068 for CPU-intensive tasks and between 0.001 and 0.711 for I/O-intensive tasks. We set the default values for the RSD parameter of long-term performance changes to the third quartile of these distributions, i. e., to 0.054 for CPU performance and 0.033 for I/O performance. Similarly, we set the default RSD value for the noise parameter to the first quartile, i. e., to 0.028 for CPU and 0.007 for I/O (see Table 3.1).

3.3.4 Stragglers and Failures

In massively distributed computational infrastructure, fault-tolerant design becomes increasingly important (Schroeder and Gibson, 2006). For the purpose of simulating fault-tolerant approaches to scheduling, DynamicCloudSim introduces straggler virtual machines and failures (SAF) during task execution. Stragglers are virtual machines exhibiting constantly poor performance (Zaharia et al., 2008). In DynamicCloudSim, the probability of a virtual machine being a straggler can be specified by the user along with the coefficient that determines how much the performance of a straggler is diminished.

We propose default values of 0.015 and 0.5, respectively, for the straggler likelihood and performance coefficient parameters. These values are based on the findings of Zaharia et al. (2008), who encountered three stragglers with performance diminished by 50 % or more among 200 provisioned virtual machines in their experiments. The numbers are backed up by the observations of Zhang et al. (2014), who encountered an amount of stragglers below 5 % in an experiment in which virtual machines were allocated on a 160

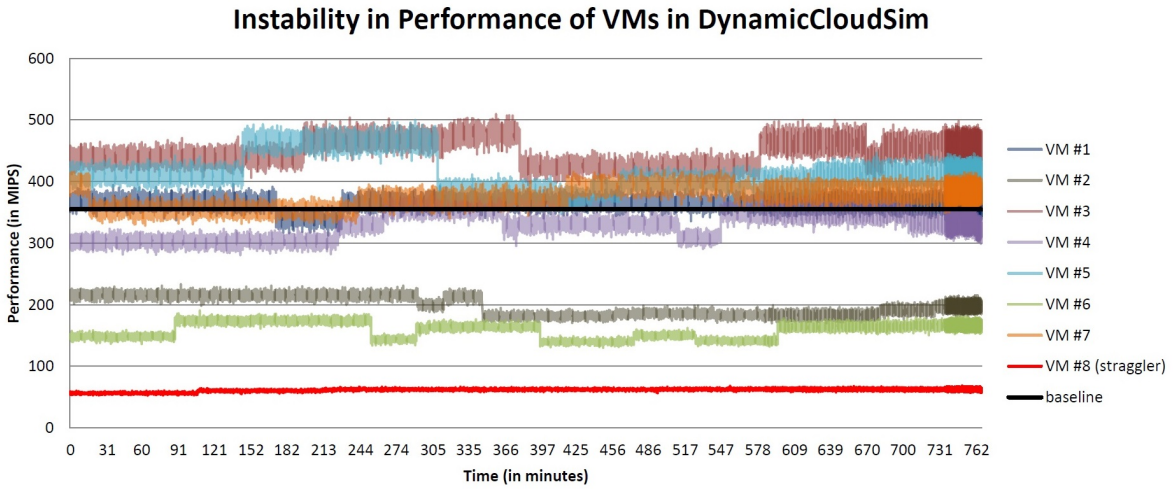


Figure 3.4: Simulated CPU performance of eight virtual machines (including one straggler) running for 12 hours with default parameters in DynamicCloudSim. The black line represents how the virtual machines' performance would have looked like in basic CloudSim.

node cluster. The effect of these parameters is exemplarily shown in Figure 3.4, which illustrates all of the introduced aspects of variability (HET, DCR, SAF) in combination for the CPU performance of eight virtual machines, including one straggler, in a simulation of 12 hours in DynamicCloudSim.

Failures during task execution are another factor of instability commonly encountered in distributed computing. DynamicCloudSim employs a basic method of failure generation: Whenever a task is assigned to a virtual machine and its execution time is computed, DynamicCloudSim determines whether the task is bound to succeed or fail. This decision is based on the average rate of failure specified by the user. The default value for the rate of failed task executions is set to 0.002, based on the observations of Jackson et al. (2010), who, in a series of experiments running on 50 virtual machines, had to restart every tenth run on average due to the occurrence of a failure on at least one virtual machine. This parameter setting is reinforced by the SLA of Amazon[2], which guarantees a machine uptime of 99.95 %.

We argue that a default task success rate of four times lower than the machine uptime guaranteed by Amazon is reasonable, since a failure during task execution can occur for different reasons. Such reasons include failure to retrieve data over the network, failure during machine start-up, or failure due to intermittent machine hang. Usually, such perturbations are not immediately recognized, hence resulting in severely increased runtimes. Consequently, in DynamicCloudSim the runtime of a failed task execution is determined by multiplying the task's execution time with a user-defined coefficient, which, by default, is set to 20 (see Table 3.1).

[2] `http://aws.amazon.com/ec2/sla/`

3.4 Simulating the Execution of Scientific Workflows

To simulate the execution of a scientific workflow in DynamicCloudSim, we require (i) values for the resource requirements (CPU, I/O, and network) of the workflow's tasks, (ii) a scheduling policy that determines how these tasks are placed on virtual machines during simulation, and (iii) baseline values for the resources provided by the physical machines (hosts) of the cloud datacenter. In Section 3.4.1, we cover the first two of these requirements by describing workflow input formats and scheduling policies supported by DynamicCloudSim. We address the third requirement in Section 3.4.2, where we describe how we set up a simulation environment resembling the inhomogeneous hardware configuration of Amazon EC2.

3.4.1 Input Formats and Workflow Scheduling Policies

DynamicCloudSim is able to interpret Hi-WAY trace files (see Sections 5.1 and 5.1.5 for a description of Hi-WAY and its trace files, respectively) as well as synthetic DAX workflow files (see Section 2.3.1.1) generated by the Pegasus Workflow Generator[3]. Both file formats contain information regarding the file sizes of any file-based data processed and produced by all of the workflow's tasks. In addition, synthetic DAX workflow provide each task's wall-clock runtime. Conversely, Hi-WAY can be configured to also store each task's runtime in user mode in its trace files, i. e., the actual CPU time used when executing the task.

When parsing a Hi-WAY trace file or synthetic DAX workflow, DynamicCloudSim determines the resource requirements of the workflow's tasks as follows: It interprets a task's CPU load as the task's wall-clock runtime or, when available, user-mode runtime in milliseconds. Assuming that the workflow's input and output data have to be read and written from remote sources, it sets a task's network load to the sum of file sizes of any input and output data processed and produced by the task. Similarly, it sets a task's disk I/O load to the sum of file sizes of any intermediate data processed and produced by the task (see Section 2.1 on page 8 for definitions of input, output, and intermediate data).

Since user-mode runtime provides a more reliable indicator of a task's CPU load than wall-clock runtime, we utilize Hi-WAY trace files for the experiments outlined in the next sections. To generate these traces, we ran workflows using Hi-WAY and Apache Hadoop 2.2.0 (see Section 2.2.3.2) as workflow execution engine. Workflows were executed on a single core of a Dell PowerEdge R910 with four Intel Xeon E7-4870 processors (2.4 GHz, 10 cores) and 1 TB of main memory, which served as the reference machine for performance measurements.

DynamicCloudSim supports the following workflow schedulers for simulations of workflow execution: (i) a knowledge-free round-robin scheduler, (ii) the static HEFT scheduling heuristic (Topcuoglu et al., 2002), (iii) a knowledge-free first-come-first-served (FCFS)

[3] `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`

Table 3.2: SPEC CFP® 2006 benchmark results for the processors of machines found in Amazon EC2 and on our reference machine.

machine	cores	SPEC CFP® base score	fraction of reference	URL
Intel Xeon E7-4870 2.4 GHz	10	51.0	100 %	http://tinyurl.com/d3oghak
Intel Xeon E5430 2.66 GHz	8	18.1	35.5 %	http://tinyurl.com/bckaqow
AMD Opteron 2218 2.6 GHz	4	12.6	24.7 %	http://tinyurl.com/ajqj3n3
AMD Opteron 270 2.0 GHz	4	8.89	17.4 %	http://tinyurl.com/aug9xcq

scheduler, and (iv) the adaptive LATE algorithm (Zaharia et al., 2008). All of these schedulers are described in detail in Section 2.2.4.

3.4.2 Setting up a Datacenter

In the experiments outlined in the next sections, we attempt to mirror the computational environment of Amazon EC2. Jackson et al. (2010) determined Intel Xeon E5430 2.66 GHz, AMD Opteron 270 2.0 GHz, and AMD Opteron 2218 HE 2.6 GHz as the underlying hardware of EC2 datacenters. To compare CPU performance between these machines and our reference machine (Intel Xeon E7-4870), on which we measured the tasks' CPU time, we consulted SPEC CFP® 2006 benchmark results for these machines. SPEC CFP® 2006 is the floating point component of the SPEC CPU® 2006 benchmark suite. It provides a measure of how fast a single-threaded task with many floating point operations is completed on one CPU core. An overview of the benchmark results for the four aforementioned types of processors is displayed in Table 3.2.

For all of the experiments outlined in the next sections, a DynamicCloudSim datacenter was initialized with 500 simulated host machines: 100 Xeon E5430, 200 Opteron 2218, and 200 Opteron 270. Since the Xeon E5430 has twice as many cores as the AMD machines, each type of machine contributes to the datacenter with an equal amount of cores and thus compute units. The simulated CPU performance of each core of these machines was set to the ratio of the machine's SPEC CFP® 2006 score to the reference machine's score. For instance, the CPU performance of Xeon E5430 machines was set to 355, yielding a runtime of 28,169 milliseconds for a task that took 10,000 milliseconds on the Xeon E7-4870 reference machine.

We assume input and output data of workflows to be stored remotely and intermediate data of workflows to be placed on shared storage such as HDFS or Amazon S3. Different measurements of network throughput within Amazon EC2 and S3 ranging from 10 to 60 MB/s have been reported (Garfinkel, 2007; Jackson et al., 2010; Pelletingeas, 2010). We therefore set the default I/O throughput of host machines to 20 MB/s. The network throughput of host machines was set to 0.25 MB/s, based on the remote access performance of S3 reported by Palankar et al. (2008) and Pelletingeas (2010).

3.5 Validation on Amazon EC2

To validate DynamicCloudSim's ability to simulate a real cloud infrastructure, we compared the simulated execution of a scientific workflow in DynamicCloudSim against actual runs on Amazon EC2. We outline the setup of the evaluation experiment in Section 3.5.1 and present results in Section 3.5.2.

3.5.1 Methods

In this evaluation experiment, we employed a Montage workflow that builds a one square degree mosaic of the Omega nebula (see Section 2.1.2). This workflow consisted of 387 tasks reading and writing 7.3 GB of data of which 128 MB are input and output files.

To conduct simulations of this workflow in DynamicCloudSim, we first generated a Hi-WAY execution trace of this workflow as described in Section 3.4.1. We then simulated the execution of the workflow 20 times for each of the schedulers listed in Section 3.4.1. Every simulation run was performed on a set of eight virtual machines with one compute unit and 1.7 GB of main memory each. We used the datacenter configuration described in Section 3.4.2 as well as DynamicCloudSim's default variability and instability parameters as determined and presented in Sections 3.3.2–3.3.4.

Similar to the simulations in DynamicCloudSim, we also executed the Montage workflow 20 times for each scheduler using Hi-WAY and Hadoop 2.2.0 on clusters of virtual machines in Amazon EC2. These clusters were spread evenly across the EC2 datacenters of US East (Virginia), US West (California), and Europe (Ireland). Cluster allocations and workflow executions were also spread across different times of day. The clusters consisted of virtual machines of type m1.small with one EC2 compute unit, 1.7 GB RAM, and 8 GB elastic block storage each. The operating system on the virtual machines was a 64-bit Ubuntu Server 12.04.3 LTS.

There were four noteworthy differences between the simulations in DynamicCloudSim and the actual executions on EC2:

1. The HEFT scheduler requires runtime estimates for each task on each machine. To provide Hi-WAY with these estimates, we executed each task once on each machine prior to workflow execution, measuring wall-clock runtime in the process. Figure 3.5 shows these measured execution times, indicating substantial heterogeneity across compute nodes.

2. The LATE scheduler requires progress estimates for each running task. Due to the black-box property of scientific workflow tasks, progress estimation is infeasible for scientific workflows in practice. To account for the lack of progress estimates in a real-world scenario, we distorted LATE's progress estimates in DynamicCloudSim by sampling from a normal distribution.

   To compare against LATE scheduling with distorted progress estimates, we implemented a more naive task replication strategy in Hi-WAY, which is related to LATE, yet does not require progress estimates. In this naive replication strategy, tasks

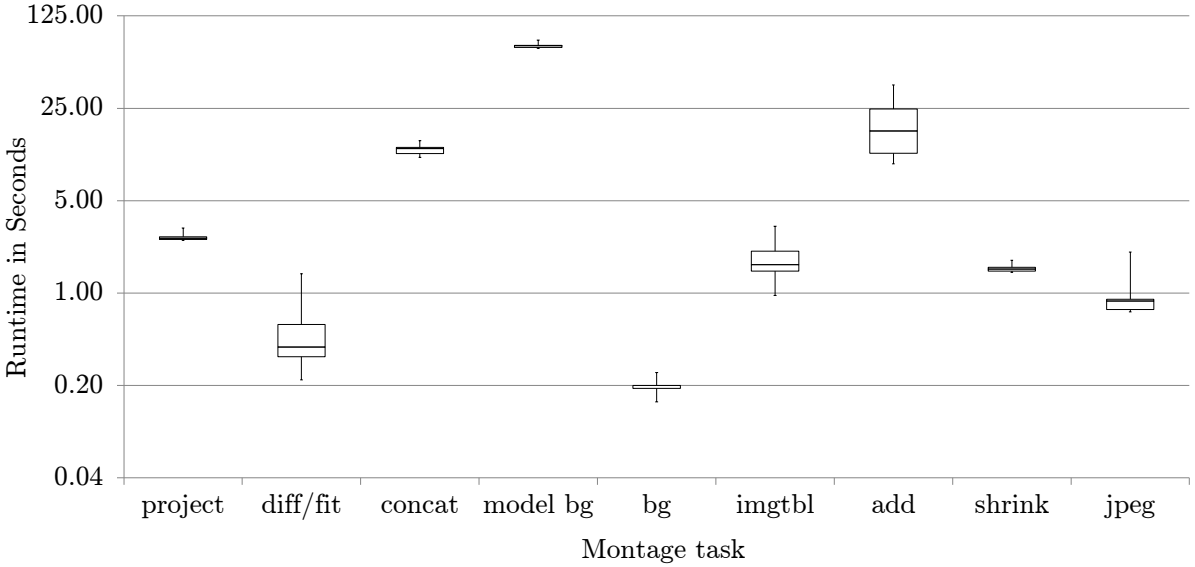


Figure 3.5: Runtimes (in log scale) of every Montage task occurring in the evaluation workflow, executed once per provisioned EC2 compute node.

are selected for speculative replication at random whenever there are idle virtual machines during workflow execution.

3. The clusters on Amazon EC2 comprised nine virtual machines. Similar to the simulation experiments in DynamicCloudSim, eight instances served as worker machines. However, an additional ninth instance served as the central master for Hi-WAY's and Hadoop's master processes (RM, NN, and AM).

4. To be able to compare workflow runtime between Hi-WAY and DynamicCloudSim, the overhead introduced by Hi-WAY (initialization of a workflow run, initialization of worker containers, etc.) was deducted from runtime measurements.

We expect the observed behavior of the four workflow schedulers to be similar in simulation and on real cloud infrastructure. Furthermore, we expect the workflow execution times on EC2 to be comparable to the execution times of simulated runs in DynamicCloudSim with default parameters.

3.5.2 Results and Discussion

Table 3.3 contrasts measured workflow execution times between DynamicCloudSim and Amazon EC2. Specifically, it lists the mean and standard deviation values for the different workflow schedulers in both settings. See also Figure 3.6 for box plots of the measured runtimes.

Slightly different mean workflow execution runtimes across all four schedulers were observed overall between DynamicCloudSim and EC2. However, using a two-tailed t-test, this difference was not found to be significant (p-value: 0.187). When compared

Table 3.3: Mean values and standard deviations of Montage workflow execution runtimes in DynamicCloudSim as opposed to on Amazon EC2.

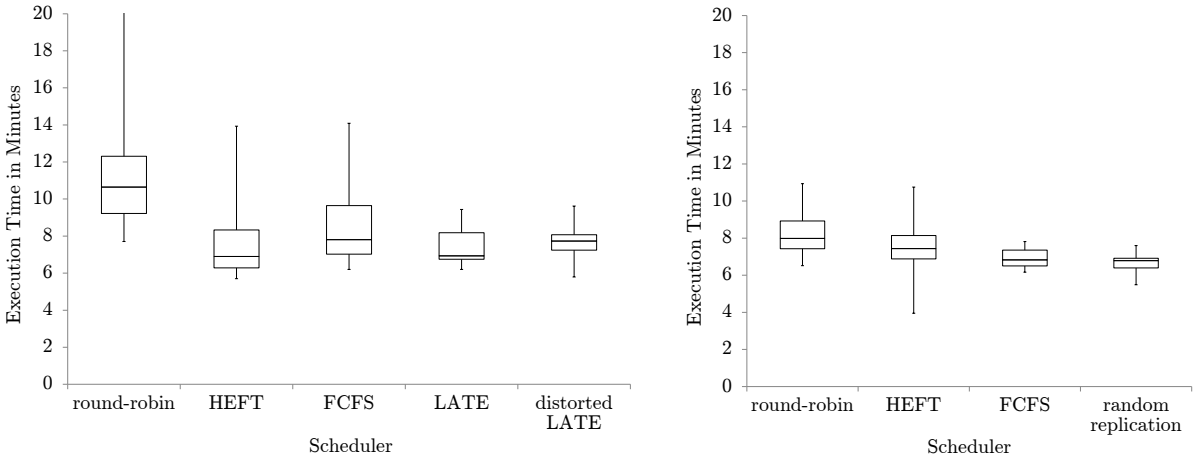
	DynamicCloudSim		Amazon EC2	
configuration	mean	STD	mean	STD
round-robin	16.166 min	16.674	8.194 min	1.166
HEFT	7.899 min	2.493	7.23 min	1.585
FCFS	8.754 min	2.423	6.905 min	0.489
LATE	7.485 min	1.080		
distorted LATE / random replication	7.721 min	0.898	6.592 min	0.517

to the workflow executions on Amazon EC2, a substantially higher variance in workflow runtime was observed in DynamicCloudSim for round-robin and FCFS scheduling. We attribute this finding to the appearance of stragglers and failures in DynamicCloudSim, which we did not encounter to a similar extent during our experiments on Amazon EC2. The lack of observed stragglers and failures on EC2 also contributes to the higher average runtime of the workflow in DynamicCloudSim when using round-robin scheduling, since this scheduler is particularly bad at handling stragglers and failures.

In DynamicCloudSim, both the HEFT scheduling heuristic and FCFS scheduling performed significantly better than the baseline round-robin scheduler (p-values: 0.017 and 0.028). Furthermore, the LATE scheduler not only significantly outperformed FCFS scheduling (p-value: 0.019), but also exhibited less variance in workflow runtimes. Notably, heavy distortion of the progress estimates utilized by the LATE scheduler did not have a major impact on the scheduler's performance. Apparently, the selection strategy for speculative task replication is not essential for increasing robustness against instability.

Very similar observations were made when executing the workflow on Amazon EC2: Both HEFT and FCFS scheduling provided significant improvements on workflow execution times when compared to knowledge-free round-robin scheduling (p-values: 0.017 and $2.6 \cdot 10^{-5}$ respectively). Furthermore, the random replication strategy provided significant runtime improvements over FCFS scheduling (p-value: 0.045).

Over the course of the experiment, we observed that both the simulation runs and the actual workflow executions on EC2 provide similar answers regarding the strengths and weaknesses of the four investigated workflow schedulers as well as recommendations of which scheduler to utilize on a computational cloud. While the differences in observed variance might warrant a slight re-weighting of DynamicCloudSim's default parameters, we find that DynamicCloudSim provides an adequate model of the variability and instability encountered in computational clouds like Amazon EC2.



(a) The runtimes of simulated Montage executions in DynamicCloudSim using default parameters.

(b) The runtimes of actual executions on Amazon EC2.

Figure 3.6: Montage execution runtimes in DynamicCloudSim versus on Amazon EC2. Round-robin, HEFT, and FCFS schedulers were present both in simulation and on Amazon EC2 and can thus be compared against one another. We also compared LATE with heavily distorted progress estimates, in which tasks are selected for speculative execution nearly arbitrarily, against a random replication strategy on EC2.

3.6 Variability and Workflow Scheduling

As an application example for DynamicCloudSim, we analyzed the effects of instability and variability in the computational infrastructure on scientific workflow scheduling. To this end, we simulated the execution of two workflows: a Montage workflow (see Section 2.1.2) and a SNV calling workflow (see Section 2.1.1.1). We then examined how different mechanisms of scheduling – knowledge-free, static, and adaptive – perform when having to deal with increasing levels of instability in the computational infrastructure. We expect the schedulers to differ in their robustness to instability, which should be reflected in diverging workflow execution times.

3.6.1 Methods

First, we used an instance of the Montage workflow which, in contrast to the Montage workflow presented in Section 3.5.1, builds a much larger (twelve square degree) mosaic of the Omega nebula. This workflow consists of 43,318 tasks reading and writing 534 GB of data in total, of which 10 GB are input and output files which have to be uploaded to and downloaded from the computational infrastructure.

As a second evaluation workflow, we employed a SNV calling workflow that was implemented using the functional workflow language Cuneiform (Brandt et al., 2015). This workflow reads two colorectal cancer cell lines, Caco-2 and GEO, to find genomic vari-

ants specific to one or the other. Genomic reads were split into distinct files of 5 MB to employ data parallelism and achieve a high degree of parallelism (see Definition 4 on page 15). Reads were mapped against the human chromosome 22 using three different mapping tools – Bowtie (Langmead et al., 2009b), SHRiMP (David et al., 2011), and PerM (Chen et al., 2009). Other chromosomes were excluded from this analysis, since chromosomes vary substantially in size and we intended to keep the variance in runtime for tasks belonging to the same bag (see Definition 3 on page 9) low. The resulting mappings were merged using SAMtools (Li et al., 2009) and variants were detected using VarScan (Koboldt et al., 2009). This resulted in a scientific workflow comprising 4,266 tasks reading and writing 436 GB of data in total.

We generated workflow traces of these two workflows and used the workflow schedulers as described in Section 3.4.1. We then generated the datacenter in DynamicCloudSim as described in Section 3.4.2. In the course of the experiments, we incrementally raised the level of variability in the simulated computational infrastructures. We conducted four experiment runs, in which we measured the effect of heterogeneity (HET), dynamic performance changes at runtime (DCR), and straggler virtual machines and faulty task executions (SAF):

1. HET: We measured the effect of heterogeneous computational infrastructure on different approaches to workflow scheduling. To this end, the performance baseline RSD parameters responsible for inhomogeneity were incrementally set to 0, 0.125, 0.25, 0.375, and 0.5 (for CPU, I/O, and network performance). The simulation of dynamic performance changes at runtime (DCR) as well as straggler virtual machines and failed tasks (SAF) was omitted.

2. DCR: We examined how persistent changes in the computational infrastructure affect workflow scheduling. Therefore, the persistent performance change RSD parameters, which is responsible for long-term changes in the performance of a virtual machine, was varied between 0, 0.125, 0.25, 0.375, and 0.5. The average rate of performance changes was fixed at 0.5 and the RSD parameters for short-term performance fluctuations and noise were set to 0.025 across all runs. HET and SAF parameters were set to 0 in this setting.

3. SAF: We determined the effect of straggler resources and failures during task execution. For this reason, the likelihoods of a virtual machine being a straggler and of a task to fail were set to 0, 0.00625, 0.0125, 0.01875, and 0.025. The performance coefficient of straggler resources was set to 0.1 and the factor by which the runtime of a task increases in the case of a failure was set to 20. To measure the effects of stragglers and failures in isolation, HET and DCR parameters were set to 0.

4. Extreme parameters: In this setting, we determined the effect of combining all introduced aspects of variability at a very high level. To this end, we utilized 1.5 times the maximum values for heterogeneity, dynamics and stragglers / failures

from the former three experiments. This translates to RSD parameters of 0.75 for HET and DCR along with straggler and failure likelihoods of 0.0375 for SAF.

For each of these configurations, we simulated 100 executions of both the 43,318 task Montage workflow and the 4,266 task genomic sequencing workflow on eight virtual machines. Virtual machines were configured to have two compute units and 3.75 GB main memory each, resembling EC2 instances of type m1.medium.

We expect the round-robin scheduler to perform well in homogeneous and stable computational infrastructures. Adding heterogeneity (HET), dynamic changes at runtime (DCR) or stragglers and failures (SAF) to the experiment should heavily diminish its performance. In our simulation experiments in DynamicCloudSim, HEFT is provided with accurate runtime estimates based on the execution time of each task on each CloudSim virtual machine at the time of its allocation. Hence, we expect the static HEFT scheduler to perform well in both homogeneous and heterogeneous infrastructures. However, we expect poor performance if dynamic changes (DCR) or failures in the computational infrastructure (SAF) are introduced and runtime estimates become inaccurate. Due to HEFT being a static scheduler, we expect a knowledge-free FCFS scheduler to outperform HEFT when dynamic changes (DCR, SAF) in the computational infrastructure are introduced. Finally, we expect an adaptive scheduler like LATE to outperform FCFS scheduling when stragglers and failures (SAF) are prevalent, since speculative replication of tasks should heavily increase robustness.

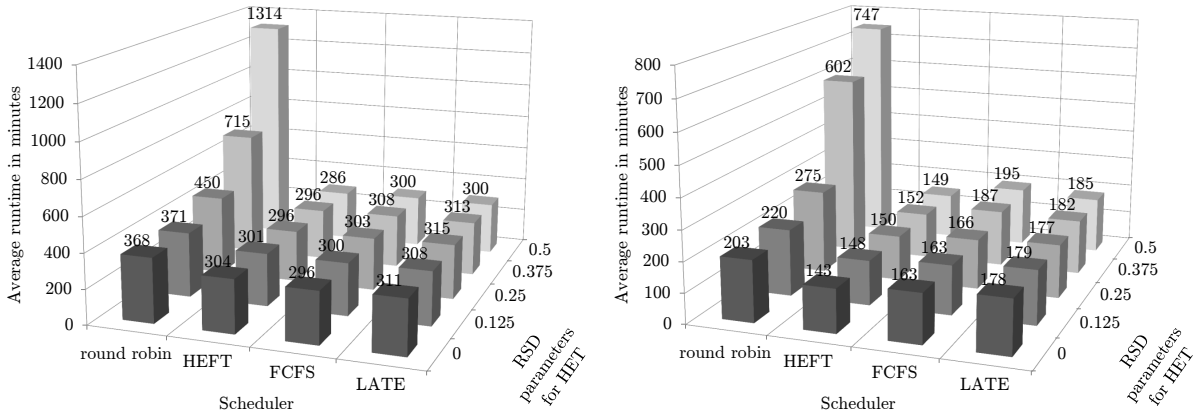
3.6.2 Results and Discussion

The results of the experiments are displayed in Figures 3.7 to 3.10. Over the course of the entire experiments, we observed average runtimes between 296 and 13,195 minutes for Montage and between 143 and 1,990 minutes for genomic sequencing. Evidently, the variability and instability simulated by DynamicCloudSim, particularly the HET and SAF parameters, have a considerable impact on execution time, especially for static schedulers.

3.6.2.1 Heterogeneity

In the experiment simulating the effect of heterogeneous resources (HET), all schedulers except round-robin exhibited robustness to even the highest levels of variance (see Figure 3.7). HEFT has been designed specifically with inhomogeneous computational resources in mind. Online schedulers like FCFS and LATE automatically assign more tasks to faster resources. Conversely, the knowledge-free round-robin scheduler is oblivious to the computational infrastructure and simply assigns an equal amount of tasks to each resource. This results in faster resources being idle while waiting for slower resources to finish.

Since LATE always reserves 10 % of the available resources for speculative replication, we would expect runtimes slightly below a greedy FCFS-based policy. However, we observed LATE to be comparable in performance to the FCFS scheduler. Also, we found



(a) Average execution times of the Montage workflow.

(b) Average execution times of the SNV calling workflow.

Figure 3.7: Effects of heterogeneity (HET) on workflow runtime using different schedulers in DynamicCloudSim.

HEFT to even have a slight edge over all other scheduling policies. We mainly attribute these findings to the presence of computationally intensive tasks blocking the execution of all successor tasks (so-called *pipeline blockers*). One such pipeline blocker is present in Montage (model background) and multiple such tasks can be found in the SNV workflow. In contrast to FCFS, HEFT and LATE are able to assign such tasks to well-suited computational resources, instead of simply assigning it to the first available resource. HEFT does this by consulting the accurate runtime estimates of all task-resource-assignments it has been provided with. LATE starts a speculative copy of the task on a compute node performing above average.

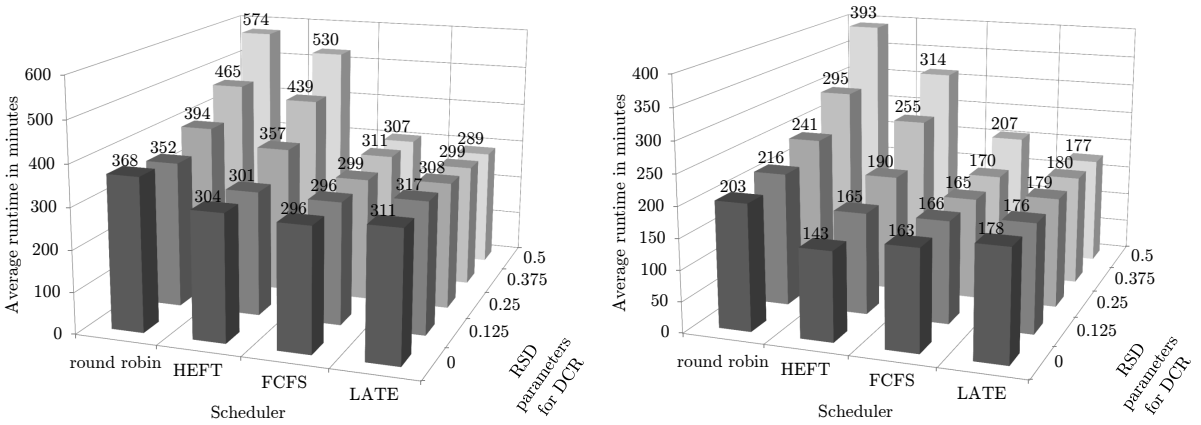
Notably, finding only the round-robin scheduler to perform subpar in this experimental setting confirmed our expectations outlined in the last section.

3.6.2.2 Dynamic Changes at Runtime

In the second part of the experiment, we examined how dynamic changes in the performance of virtual machines (DCR) affect the runtime of the evaluation workflows achieved by the four scheduling mechanisms (see Figure 3.8). The results confirm our expectations of schedulers like knowledge-free round-robin and static HEFT not being able to handle dynamic changes. The major shortcoming of these schedulers lies in the fact that they perform a fixed assignment of tasks to resources, which is strictly abided by during workflow execution. Therefore, changes in the runtime environment make even elaborate static schedules like HEFT suboptimal.

3.6.2.3 Stragglers and Failures

In the third part of the experiment, we measured how the appearance of straggler virtual machines and failed task executions (SAF) influence the performance of the four examined



(a) Average execution times of the Montage workflow.

(b) Average execution times of the SNV calling workflow.

Figure 3.8: Effects of changes of performance at runtime (DCR) on workflow execution time using different schedulers in DynamicCloudSim.

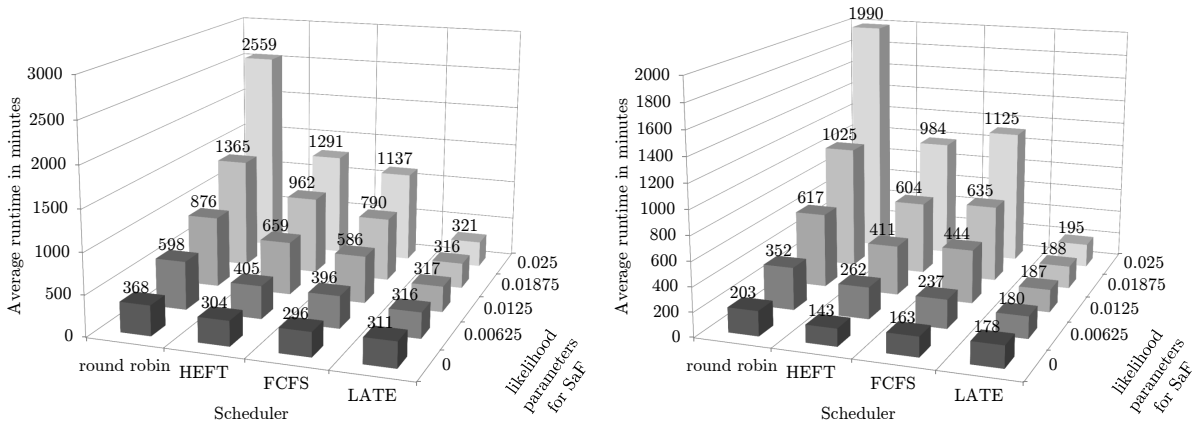
workflow schedulers. Figure 3.9 confirms the robustness of the LATE scheduler even for high amounts of failures and stragglers. Evidently, speculative replication of tasks with a low progress rate alleviates the problems introduced by stragglers and failures.

In contrast, the performance of all other schedulers diminished quickly in the face of failure: As mentioned previously, both evaluation workflows contain pipeline blockers, i. e., computationally intensive tasks that block the execution of other upstream tasks and severely limit the degree of parallelism during workflow execution. Such tasks are particularly problematic if their execution fails or if they are assigned to a straggler virtual machine. Knowledge-free schedulers like round-robin or FCFS cannot detect stragglers and will occasionally assign these critical tasks to straggler VMs, resulting in severely increased workflow execution times. Conversely, HEFT is able to exploit heterogeneity and detect straggler machines by means of the runtime estimates it is provided with. However, HEFT is unable to cope with failures during workflow execution.

Notably, the genomic sequencing workflow exhibits more pipeline blockers than the Montage workflow. For this reason, the performance degradation for high SAF values is more apparent in the genomic sequencing workflow than in the Montage workflow. For very high DCR values, virtual machine performance baseline changes can occasionally reach the extent of the virtual machine temporarily behaving like a straggler. For workflows with frequent pipeline blockers, this can lead to reduced performance of schedulers that can usually cope with DCR, as seen when executing the genomic sequencing workflow with the FCFS scheduler at DCR=0.5 (see Figure 3.8).

3.6.2.4 Extreme Variability

In the fourth part of the experiment, we examined how all three of the introduced aspects of variability combined and taken to extremely high levels (RSD parameters of 0.75 for HET and DCR; likelihood parameters of 0.0375 for SAF) influence the workflow



(a) Average execution times of the Montage workflow.

(b) Average execution times of the SNV calling workflow.

Figure 3.9: Effects of straggler virtual machines and failed tasks (SAF) on workflow runtime using different schedulers in DynamicCloudSim.

execution time. The results of this experiment for Montage are shown in Figure 3.10. Once again, LATE is the only scheduler to exhibit robustness to these extreme parameter configurations. Furthermore and in contrast to the findings in the third experiment, the HEFT scheduler substantially outperforms FCFS in such settings.

The combination of all factors, i. e., dynamic changes at runtime to inhomogeneous compute resources which can also be stragglers or subject to failed task execution, can lead to cases in which the execution of the pipeline blocker in Montage (model background) can take an extremely long time. This is more problematic for a knowledge-free scheduler such as FCFS, which assigns a task to the first available computational resource, which might be a straggler. In contrast, HEFT is at least able to handle heterogeneous and straggler resources by means of its accurate runtime estimates.

3.6.2.5 Implications for Scientific Workflow Scheduling

The last two experiments illustrated the severe effect of straggler virtual machines and failed tasks executions (SAF) on (simulated) workflow runtime, confirming previous reports on the importance of fault-tolerant design in computationally intensive applications (Schroeder and Gibson, 2006; Chen and Deelman, 2012). While the simulation was able to replicate the advertised strengths of LATE, we acknowledge that more sophisticated failure models would be a desirable enhancement over their current implementation in DynamicCloudSim.

All in all, the experiments clearly confirmed the expectations described in Section 3.6. The simulations underline the importance of adaptive scheduling of scientific workflows in shared and distributed computational infrastructures like public clouds. In particular, the experiments showcased the benefits to be gained from exploiting heterogeneous resources and speculatively replicating tasks. These insights strongly motivated the heuristics underlying the adaptive scheduler ERA, presented in the next chapter.

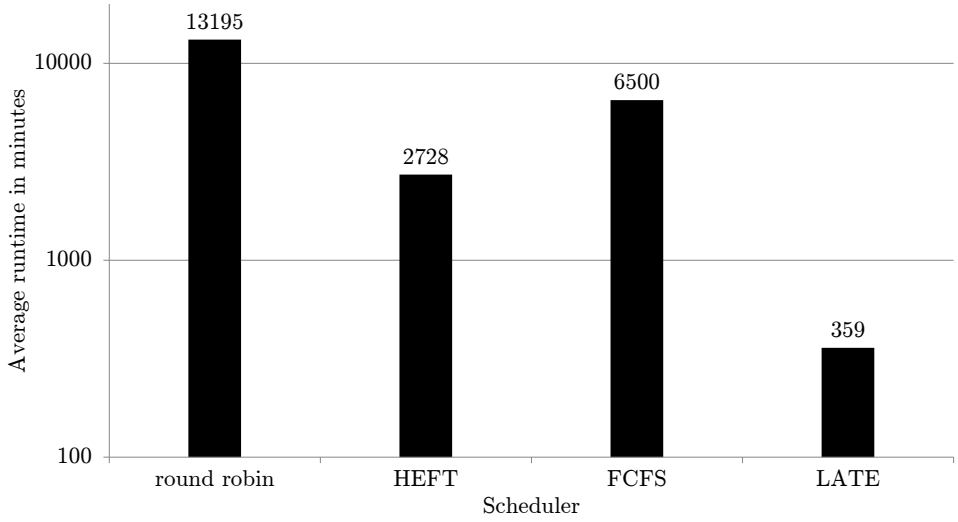


Figure 3.10: Execution time (in log scale) of the Montage workflow in DynamicCloudSim in extreme cases of instability.

Table 3.4: Features of CloudSim, WorkflowSim, and DynamicCloudSim.

Feature	CloudSim	WorkflowSim	DynamicCloudSim
performance characteristics MIPS, bandwidth, memory	✓	✓	✓
performance characteristic file I/O			✓
runtime of a task depending on values other than MIPS			✓
modeling of data dependencies	✓	✓	✓
workflow parsing		✓	✓
implementation of workflow schedulers		✓	✓
modeling of delays at different layers of a workflow system		✓	
support for task clustering		✓	
different virtual machines on different hosts	✓	✓	✓
resource allocation based on compute units			✓
dynamic changes of virtual machine performance at runtime			✓
modeling of failures during task execution		✓	(✓)
introduction of straggler virtual machines			✓

3.7 Related Work

Merdan et al. (2008) and Hirales-Carbajal et al. (2010) developed simulation environments specifically for comparing different approaches to workflow scheduling on computational grids. They also provide examples of possible experimental setups, yet omit the execution of these experiments. Our work differs from these publications in three ways: Firstly, by extending a universal simulation framework like CloudSim, DynamicCloudSim is not limited to the field of scientific workflows, but can be utilized for simulation of any cloud application. Secondly, our work puts a strong emphasis on instabilities in the computational infrastructure, which is important to achieve realistic results. Thirdly, we conduct an experimental validation of the changes added to the simulation toolkit.

Chen and Deelman (2012) presented WorkflowSim as another extension to CloudSim. WorkflowSim is tightly coupled to the workflow management system Pegasus (see Section 2.3.1.1) and adds to CloudSim (i) a model of delays occurring in the various levels of the Pegasus stack (e. g., queue delays, pre/post-processing delays, data transfer delays), (ii) an elaborate model of node failures, and (iii) the implementations of several workflow schedulers, including FCFS, HEFT, Min-Min, and Max-Min (see Section 2.2.4). Parameters are directly learned from traces of real executions. WorkflowSim has no notion of heterogeneous hardware or variance in available resources. In contrast, DynamicCloudSim directly models instability and heterogeneity in the environment in which a workflow, or any other collection of computationally intensive tasks, is executed. DynamicCloudSim is thus independent of the computational paradigm (e. g., scientific workflows) and the concrete system of execution (e. g., the workflow management system Pegasus). See Table 3.4 for a comparison of features available in CloudSim, WorkflowSim, and DynamicCloudSim.

3.8 Summary

In this chapter, we presented DynamicCloudSim as an extension to CloudSim, a popular simulator for evaluating resource allocation and scheduling strategies on distributed computational architectures. We enhanced CloudSim's model of infrastructure-as-a-service cloud computing infrastructure by introducing models for (i) inhomogeneity in the performance of computational resources, (ii) uncertainty in and dynamic changes to the performance of virtual machines, and (iii) straggler machines and failures during task execution.

We validated the models of instability introduced in DynamicCloudSim by comparing the simulated execution of a workflow in DynamicCloudSim against actual runs on Amazon EC2. Finally, we showed that introducing performance variability to scientific workflow execution using four established scheduling algorithms and two evaluation workflows replicated the known strengths and shortcomings of these schedulers. These findings underline the importance of adaptivity in scheduling of scientific workflows on shared and distributed computational infrastructures.

In the next chapter, DynamicCloudSim is employed to evaluate an adaptive workflow scheduler, which is partly inspired by the scheduling heuristics evaluated in Section 3.6.

4 Adaptive Scheduling of Scientific Workflows

Data-intensive scientific workflows usually comprise a number of heterogeneous (bags of) tasks (see Section 2.1). Similarly, the computational infrastructures required to execute such workflows in a reasonable time frame are usually composed of multiple, oftentimes heterogeneous compute nodes (see Section 2.2.2). In this section, we consider the scientific workflow scheduling problem, which we define as follows.

Definition 7 (Scientific Workflow Scheduling) *Let m be the number of available unrelated machines and $S = (D, T, E)$ a scientific workflow comprising $|T| = n$ tasks. Let $e_t^{i,j}$ be the amount of time required for the processing of task i on machine j starting at time t. The objective of the scientific workflow scheduling problem is to assign tasks to machines such that precedence constraints between tasks are satisfied and the overall workflow execution time (makespan) is minimized.*

The scientific workflow scheduling problem is NP-complete in the general case (Garey and Johnson, 1979). It is also NP-complete for several restricted cases, e. g., for the case where $m = 1$ and $e_t^{i,j} \in \{1, 2\}$ (Ullman, 1975). Therefore, unless P=NP, polynomial algorithms can only find approximate solutions to the problem. For this reason, scientific workflow scheduling is typically approached heuristically in practice.

Lenstra et al. (1990) showed that, even in the absence of precedence constraints, no polynomial algorithm can consistently construct schedules with makespans less than $\frac{3}{2}$ of the optimum. However, they presented a polynomial algorithm that assembles schedules guaranteeing makespans within two times the optimum. Other algorithms with comparable guarantees have been published that take into account precedence constraints such as those observed in scientific workflows (Kumar et al., 2009; Benoit et al., 2013).

The scientific workflow community has published a number of scheduling policies that do not provide any formal guarantees, but have been shown to work well in practice (Yu et al., 2008; Alkhanak et al., 2016). Most of these scheduling algorithms have in common that (i) they exploit the heterogeneity in both the tasks comprising the scientific workflow as well as in the underlying computational infrastructure (Iverson et al., 1999) and (ii) they rely on the existence of accurate estimates $\hat{e}_t^{i,j}$ for task runtimes $e_t^{i,j}$. In real-world scenarios, the latter requirement is problematic, since the tasks comprising a scientific workflow are black boxes. Therefore, the only means of reliably estimating task runtime is to infer it from historical runtime measurements (da Silva et al., 2015). This approach is particularly feasible for bag-of-tasks workflows (see Definition 3 on

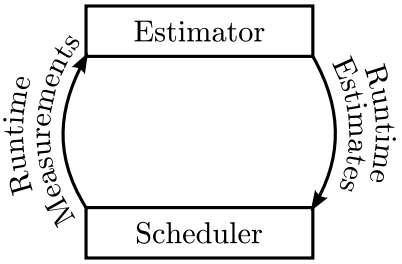


Figure 4.1: Interplay of workflow scheduling and performance estimation, traditionally viewed and addressed as two different problems.

page 9), in which measured task runtimes can be employed to estimate the runtimes of tasks belonging to the same bag. Although a considerable number of approaches towards black-box performance prediction in distributed environments have been proposed, these approaches usually attack the prediction problem in isolation of the scheduling problem.

The interplay between performance prediction and workflow scheduling is therefore as follows: The performance prediction component generates the input for the scheduling component, which, in turn, generates the input for the prediction component (see Figure 4.1). This input for the prediction component is provided in the form of new runtime measurements, which ideally are a by-product of executing tasks on their assigned machines. The caveat here is that the scheduler is likely to only assign tasks to machines if their runtime is estimated to be favorable. Consequently, no new runtime measurements will be gathered and, thus, runtime estimates will become outdated for task-machine-pairs deemed unfavorable by the scheduler.

While this interplay might work well when applied on infrastructures which are stable in size and performance, it is problematic if performance is subject to variability, which is the case for modern distributed infrastructure (see Sections 2.2.2 and 3.1). Here, a best-effort scheduler striving exclusively for short-term reductions in workflow execution times will fail to re-evaluate task-machine-assignments previously observed to be unfavorable (e. g., due to isolated heavy load on the machine in question). However, in a dynamically changing environment, a suboptimal task-machine-assignment might not stay suboptimal for long. For illustrative purposes, consider the following example.

Example *Suppose a best-effort scheduler is utilized to execute a bag-of-tasks workflow comprising two bags of tasks* $B_v = \{v_1, v_2, \ldots, v_p\}$, $B_w = \{w_1, w_2, \ldots, w_q\}$ *with no data dependencies in between. The computational infrastructure comprises two machines* x, y. *At time* t_0 *the scheduler is provided with accurate runtime estimates* $\hat{e}^{v,x}_{t_0} = e^{v,x}_{t_0} = 3$, $\hat{e}^{v,y}_{t_0} = e^{v,y}_{t_0} = 2$, $\hat{e}^{w,x}_{t_0} = e^{w,x}_{t_0} = 2$, $\hat{e}^{w,y}_{t_0} = e^{w,y}_{t_0} = 2$. *To minimize the workflow's makespan, the scheduler assigns tasks* $v_i \in B_v$ *to machine* y *and tasks* $w_j \in B_w$ *to machine* x *as long as both bags still contain ready tasks. Now suppose an unforeseeable change in the performance of machine* x *occurs at time* t_{10}, *altering the runtime of tasks* $v_i \in B_v$ *on that machine to* $e^{v,x}_{t_{10}} = 1$. *The performance estimation component is unaware of this change until another task* $v_i \in B_v$ *has been assigned to and executed on machine* x. *Therefore, the scheduler is provided with the runtime estimate* $\hat{e}^{v,x}_{t_{10}} = 3$, *based on which*

it continues to assign tasks $v \in B_v$ to machine y. It thereby prevents the generation of new runtime measurements of tasks $v_i \in B_v$ on machine x. However, such runtime measurements would be necessary to register the improvement in runtime and update the runtime estimate $\hat{e}^{v,x}_{t_{10}}$.

We argue that an adaptive scheduling mechanism for scientific workflows, i. e., a scheduler able to adapt workflow execution to a dynamically changing computational environment, not only has to make use of continuously updated runtime estimates, but must actively participate in the determination and maintenance of up-to-date runtime estimates for any task on any machine. To this end, an adaptive scheduler has to implement runtime estimation and task scheduling as an integrated component. Furthermore, it has to balance short-term reductions in a workflow's expected makespan against the necessity of capturing and preserving a comprehensive picture of up-to-date runtime measurements.

In this chapter, we present ERA, an adaptive scheduler that (i) is able to exploit heterogeneity, similar to established approaches like HEFT (see Section 2.2.4.2), (ii) replicates straggler tasks to increase robustness in the face of instability, similar to methods like LATE (see Section 2.2.4.3), and (iii) provides a mechanism to occasionally revisit task-machine-assignments for which runtime estimates are likely to be outdated. This latter heuristic ensures that all runtime estimates stay up to date, which, as discussed previously, is important on infrastructures that are subject to performance variability.

The remainder of this chapter is structured in the following way. Section 4.1 describes how ERA models task runtimes using Wiener process models. Subsequently, Section 4.2 outlines how ERA derives scheduling decisions based on these stochastic models. An evaluation of the heuristics implemented by ERA is given in Section 4.3 and performance is compared against the workflow schedulers discussed in Section 3.6. An exhaustive analysis of the related work on runtime estimation and adaptive scheduling of scientific workflows is presented in Section 4.4. Finally, a summary of our findings is given in Section 4.5.

4.1 Wiener Process Models

In this section, we describe a novel method of task runtime estimation, in which we model the runtime of a task on a machine as a stochastic process, i. e., a sequential collection of random variables. The idea of modeling a task's execution time on a given machine as a random variable with the aim of enabling adaptive scheduling mechanisms of DAG-shaped programs on heterogeneous computational infrastructure has, to the best of our knowledge, first been proposed by Iverson et al. (1999). Based on historical runtime measurements, the authors propose to build models for estimating the execution duration of any task on any machine in the distributed infrastructure. Since these models are assembled by means of regression, neither the actuality nor the temporal sequence of historical measurements is considered.

However, in the face of instability and dynamic performance changes in the computational infrastructure, runtime measurements may become outdated. Wolski et al. (2000)

therefore argue that performance estimation based on historical measurements not only requires some of these measurements to be up-to-date, but also has to interpret these measurements as time series to adequately model the temporal sequence and recency of events. Furthermore, they find that time series of performance measurements are often autocorrelated, i. e., correlated with a delayed copy of itself, and self-similar, i. e., similar to itself at different scales. The latter observation in particular is often indicative of a chaotic system and is typically encountered in Brownian motions (Embrechts and Maejima, 2000).

The term Brownian motion refers to the random movement of particles in a fluid, which was first observed by botanist Robert Brown in 1827. A Brownian motion can be modeled as a Wiener process, which is a continuous-time stochastic process named after the mathematician Norbert Wiener.

Definition 8 (Wiener Process) *A Wiener process W is is a continuous-time stochastic process, i. e., a collection of random variables $(W_t)_{t\in\mathbb{R}_0^+}$. Differences $W_t - W_s$ between subsequent random variables W_s, W_t with $0 \leq s < t$ are called increments. W is characterized by three key properties:*

1. *W almost surely starts at zero, i. e., W starts at zero with a probability of one even though the set of possible exceptions may be non-empty:*

$$W_0 = 0$$

2. *Increments $W_t - W_s$ are normally distributed and stationary with mean 0 and variance $t - s$:*

$$\forall\ 0 \leq s < t : W_t - W_s \sim \mathcal{N}(0, t - s)$$

3. *Increments of W are independent:*

$$\forall\ 0 \leq s < t \leq u < v : W_v - W_u \perp W_t - W_s$$

A Wiener process can be shifted by a constant λ along the ordinate axis and generalized to include a volatility σ. This results in a stochastic process $\widehat{W}$ with $\widehat{W}_t = \sigma W_t + \lambda$, in which increments $\widehat{W}_t - \widehat{W}_s$ are normally distributed with mean 0 and variance $\sigma^2(t - s)$.

4.1.1 Modeling Task Runtime as Wiener Process Model

We model the execution duration of any task on any machine using a generalized Wiener process model $\widehat{W}$. We estimate the parameters of $\widehat{W}$ based on historical runtime measurements. To this end, we make several assumptions:

1. We assume that observed differences (increments) in consecutive runtime measurements follow a normal distribution. To verify this assumption, we performed a small sample experiment. We executed a sequence mapping task using the tool Bowtie 2 (see Section 2.1.1.1) 200 times in succession on the same machine. We

displayed the differences in subsequent execution durations in a normal quantile-quantile plot as well as in a density plot (see Figures 4.2 and 4.3, respectively). Subsequently, we performed a Shapiro-Wilk normality test, in which we could not reject the null hypothesis of the increments being normally distributed (p-value of 0.56).

2. We assume that differences in subsequent task runtime measurements are statistically independent, an assumption commonly made in task runtime estimation (e. g., Tang et al., 2011).

3. We assume that task runtime measurements are accurate.

Let $e_{t_1}, e_{t_2}, \ldots, e_{t_n}$ be runtime measurements of the to-be-modeled task-machine-pair taken at time $t_1, t_2, \ldots, t_n$. The values for random variables $\widehat{W}_{t_i}$ can be directly inferred.

$$\widehat{W}_{t_i} = e_{t_i} \text{ for } 1 \leq i \leq n$$

Consecutive increments $\widehat{W}_{t_i} - \widehat{W}_{t_{i-1}}$ have a variance of $\sigma^2(t_i - t_{i-1})$ and, thus, a standard deviation of $\sigma\sqrt{t_i - t_{i-1}}$. Therefore, a first step in estimating the volatility σ of $\widehat{W}$ is to normalize observed differences in runtime $e_{t_i} - e_{t_{i-1}}$ to a uniform time frame. Since the standard deviation $\sigma\sqrt{t_i - t_{i-1}}$ has the same unit as the observed differences in runtime, this is achieved by dividing $e_{t_i} - e_{t_{i-1}}$ by $\sqrt{t_i - t_{i-1}}$.

$$d_{t_i} = \frac{e_{t_i} - e_{t_{i-1}}}{\sqrt{t_i - t_{i-1}}} \text{ for } 2 \leq i \leq n$$

Based on these normalized observed differences in runtime d_{t_i}, we compute the average normalized difference $\overline{d}$.

$$\overline{d} = \frac{1}{n-1} \sum_{i=2}^{n} d_{t_i}$$

Next, we determine the corrected sampling standard deviation s.

$$s = \sqrt{\frac{1}{n-2} \sum_{i=2}^{n} (d_{t_i} - \overline{d})^2}$$

Since we normalized the measured differences d_{t_i} to a uniform time frame, the corrected sampling standard deviation s gives an estimate for the volatility σ of $\widehat{W}$.

$$\sigma \coloneqq s$$

$$\forall\ 0 \leq s < t : \widehat{W}_t - \widehat{W}_s \sim \mathcal{N}(0, \sigma^2(t-s))$$

Finally, since increments are assumed to be statistically independent, the expected value of $\widehat{W}$ at the current time $t_c \geq t_n$ only depends on the latest measurement e_{t_n}.

$$\widehat{W}_{t_c} \sim \mathcal{N}(e_{t_n}, \sigma^2(t_c - t_n))$$

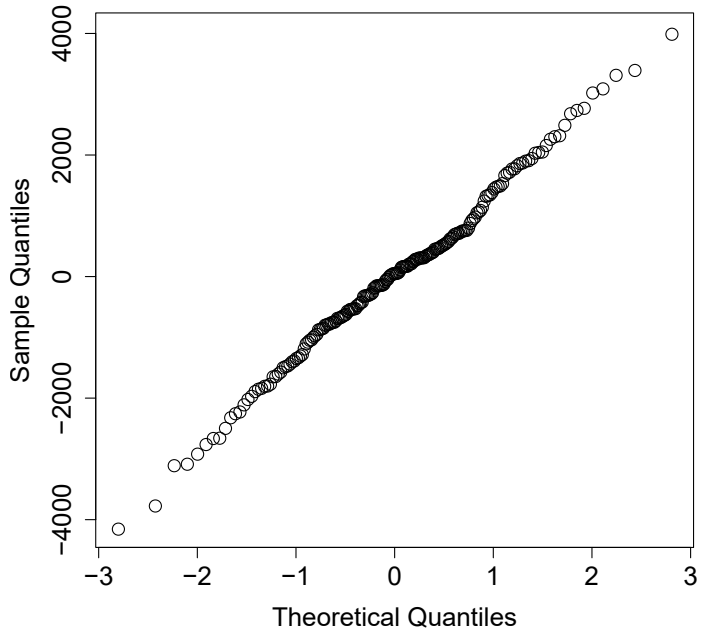


Figure 4.2: Q-Q (quantile-quantile) plot for the observed differences in measured runtimes of subsequent Bowtie 2 task invocations. The measured (sample) quantiles are plotted against the theoretical quantiles of a normal distribution. The points in the plot approximately lying on the line $y = x$ indicate the differences (increments) being normally distributed.

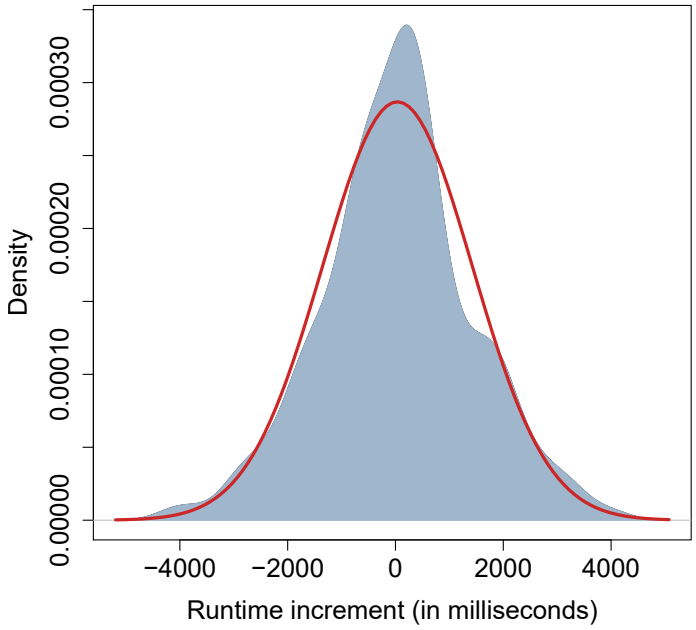


Figure 4.3: Density plot for the observed differences in measured runtimes of subsequent Bowtie 2 task invocations. The density plot (gray area) visualizes the frequency distribution of observed differences over a continuous interval and bears resemblance to a normal distribution (red line).

$$E(\widehat{W}_{t_c}) = e_{t_n}$$

The random variable $\widehat{W}_{t_c}$ may be employed to infer a current runtime estimate (see next section). Evidently, as the time since the last measurement progresses, i. e., as $t_c - t_n$ increases, the expected value of $\widehat{W}_{t_c}$ stays constant at e_{t_n}, whereas its standard deviation $\sigma\sqrt{(t_c - t_n)}$ increases. In other words, the Gaussian bell curve describing the random variable $\widehat{W}_{t_c}$ becomes wider as historical runtime measurements become increasingly outdated. This growth of uncertainty over time is also displayed in Figure 4.4.

4.2 ERA

In this section, we describe ERA, an adaptive scheduler for scientific workflows that integrates scheduling heuristics with runtime estimation based on Wiener process models. While ERA is able to schedule any scientific workflow in any computational environment, it is best-suited for scenarios in which the following conditions are met:

1. To-be-scheduled workflows are either bag-of-tasks workflows (see Definition 3 on page 9) or are executed multiple times (e. g., as a consequence of new data becoming available or an iterative approach to workflow development and refinement). Furthermore, the runtimes of tasks within the same bag, executed under the same conditions (e. g., machine, current performance), are comparable to one another. This entails that all tasks within the same bag have similar-sized input data, which, for instance, is common for data-parallel workflows comprising numerous similar tasks processing different equal-sized fragments of input data (see Section 2.2.1.2). Notably however, this assumption might not hold for all types of tasks. If the size of input data varies between tasks of the same bag, runtime estimation has to incorporate the size of input data.

2. The available machines are distributed and heterogeneous (HET) in the amount and type of resources they provide. They are also subject to dynamic changes of performance of runtime (DCR). Finally, they comprise occasional stragglers and are subject to failures (SAF), for instance as a consequence of being shared between multiple users (see Section 3.1 for a description of the aspects of variability HET, DCR, and SAF).

3. Workflows are composed of (bags of) tasks that are also heterogeneous in the amount and type of resources they require. For instance, in the case of Montage (see Section 2.1.2) the *model background* task is CPU-bound, whereas *diff* tasks leave the CPU mostly idle, straining the hard disk instead. Conversely, if the computational infrastructure is heterogeneous while the to-be-executed tasks are homogeneous across all bags of tasks, no speed-ups can be gained by determining suitable task-machine-assignments.

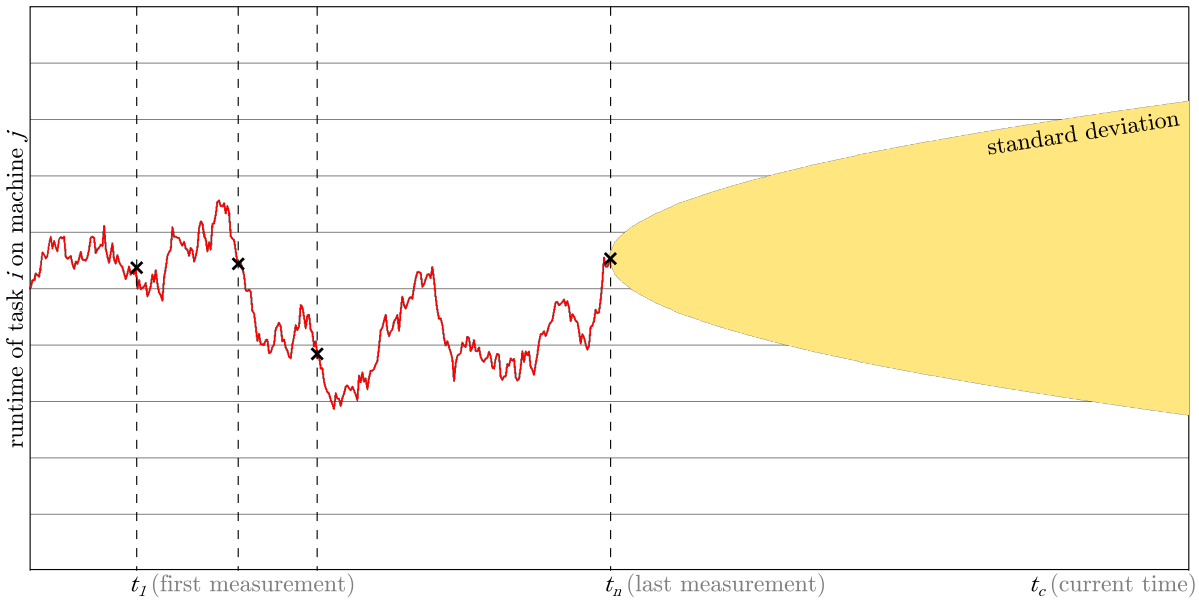


Figure 4.4: The runtime of a task i on a machine j, modeled as a generalized Wiener process model (red line). As the time since the last measurement progresses, the uncertainty, as represented by the standard deviation, increases.

4. The number of available machines is much smaller than the degree of parallelism of the workflow (see Definition 4 on page 15) throughout most stages of workflow execution.

5. Any task can be executed on any machine. This entails that sufficient resources (e. g., memory, local storage) for task execution are available and any potential software dependencies are resolved.

6. Machines are configured to run a maximum of one task at any given time. ERA does not explicitly model task collocation and contention for resources shared between tasks. Multiple tasks running on the same machine at the same time violates our earlier assumption of task runtime measurements being statistically independent and is likely to diminish scheduling performance.

We found these assumptions to hold for many scenarios of executing data-intensive scientific workflows, such as the ones presented in Sections 2.1.1 and 2.1.2, on infrastructure-as-a-service clouds where virtual machines can be tailored to the application's needs. Based on these assumptions and in order to flexibly adapt to instability in the computational infrastructure, ERA employs the following mode of operation: Whenever a machine is able to accept a new task, usually after concluding execution of a previous task, ERA chooses a new task to be executed on that machine. This new task is selected from the set of ready tasks (see page 9 for definitions of blocked, ready, running, and completed tasks). Hence, ERA is a delay-based just-in-time scheduler, as it delays task scheduling until resources are actually available (similar to, e. g., Cai et al., 2017).

The strategy for selecting a task is based on three key heuristics, which ERA balances against one another:

1. *Adaptation*: To notice and, consequently, adapt to improvement and degradation in the performance of the available distributed machines (DCR), ERA establishes and sustains a comprehensive picture of up-to-date runtime measurements for all task-machine-pairs (see Section 4.2.1).

2. *Replication*: To provide robustness to straggler machines and failures during task execution (SAF), ERA speculatively replicates the execution of critical tasks (see Section 4.2.2).

3. *Exploitation*: To determine suitable task-machine-assignments, ERA exploits the heterogeneity present both in the tasks comprising a scientific workflow as well as the underlying distributed computational infrastructure (HET) (see Section 4.2.3).

In the remaining parts of this chapter, we assume scientific workflows scheduled by ERA to be bag-of-tasks workflows (see Definition 3 on page 9). Note that any scientific workflow $S = (D, T, E)$ can be interpreted as a bag-of-tasks workflow by introducing a separate bag B_i for each task $t_i \in T$.

ERA maintains separate generalized Wiener process models for the task runtimes of any combination of bag of tasks and machines. The model $\widehat{W}^{i,j}$ specific to tasks belonging to bag B_i executed on machine j is updated whenever a new runtime measurement has been obtained for this bag-of-tasks-machine-pair. To improve the estimation of volatility parameters $\sigma^{i,j}$, ERA can also incorporate task runtime measurements from earlier executions of the same or similar workflows (i. e., workflows comprising tasks that invoke the same programs and can thus be interpreted as belonging to the same bags of tasks). In subsequent sections, we outline how ERA realizes the aforementioned concepts of adaptation, replication, and exploitation using these Wiener process models $\widehat{W}^{i,j}$. See also Algorithm 1 for a pseudocode description of the ERA scheduler.

4.2.1 Adaptation

Adaptation, i. e., the ability to maintain up-to-date runtime estimates in the face of dynamic changes of performance at runtime (DCR), is implemented twofold in ERA. First, to determine the corrected sampling standard deviation $s^{i,j}$ for a bag-of-tasks-machine-pair (B_i, j), at least three measurements of execution duration are required. ERA assumes a runtime of 0 for the execution of tasks belonging to bag B_i on machine j if sufficient measurements are not yet available for this configuration. Since a task with runtime 0 is highly likely to be selected for execution, ERA thereby guarantees that sufficient measurements are collected swiftly.

Secondly, since increments are assumed to be independent, the expected execution duration of a task of bag B_i on machine j at the current time t_c is equal to the latest runtime measurement $e^{i,j}_{t_n}$ taken at time t_n. However, ERA sets the runtime estimate $\hat{e}^{i,j}_{t_c}$ for tasks of bag B_i on machine j not to $\widehat{W}^{i,j}_{t_c}$'s expected value, but instead to the value of

its α-quantile, with $0 < \alpha \leq 0.5$ being an adjustable parameter. Note that the standard deviation of $\widehat{W}_{t_c}^{i,j}$ increases as time progresses (see last section). Hence, the more time has passed since the last measurement $e_{t_n}^{i,j}$ for a task of bag B_i on machine j was taken and the lower the value for α is, the lower of an estimate $\hat{e}_{t_c}^{i,j}$ is provided by ERA. In other words, ERA becomes increasingly optimistic about the execution duration of tasks of bag B_i on machine j if such tasks have not been run on j for a long time. Therefore, through its α-parameter, ERA allows to balance the need for up-to-date runtime measurements against the accuracy of runtime estimation.

Note that if α is set to 0.5, the adaptivity heuristic is effectively disabled. For other settings of α however, a runtime estimate provided by ERA could theoretically become negative if (i) volatility is large, (ii) a large amount of time has passed since the last runtime measurement, and/or (iii) very small values for α are chosen. While we have never observed such behavior in practical applications, ERA circumvents this issue by never providing a runtime estimate below one millisecond. Alternatively, ERA can be configured to operate on logarithmized runtime measurements and exponentiate runtime estimates[1].

4.2.2 Replication

Replication of running tasks has been integrated as the central heuristic of the LATE scheduler (see Section 2.2.4.3) to deal with stragglers and failures (SAF). In our experiments on DynamicCloudSim (in Section 3.6.2), we found this heuristic to provide promising results especially for workflows with pipeline blockers, i. e., computationally intensive tasks that block the execution of other upstream tasks.

ERA implements a replication heuristic similar to LATE. Specifically, when a workflow execution reaches a point where a machine is able to execute further tasks, yet there are no more unassigned ready tasks available, ERA will instead assign a replicate of an already running task to that machine. Thus, speculative replication is only enabled when resources are idle. The maximum number of concurrently running replicate tasks can be configured via ERA's ρ parameter.

The task selection strategy for speculative replication is similar to that of regular task execution (see next section). However, the adaptation heuristic is disabled here (i. e., α is set to 0.5). The rationale behind this is that replicated tasks are often critical to workflow execution and, thus, should be assigned to the most suitable machines. This assures that other idle machines can commence execution of blocked upstream tasks as early as possible.

Similarly to LATE, when an instance of a task (i. e., its original or one of its replicates) finishes execution, ERA terminates all other instances of that task.

[1]Note that this mode of operation would model task runtime not as a Wiener Process / Brownian Motion, but as a Geometric Brownian Motion (Ross, 2014).

4.2.3 Exploitation

Considering the heterogeneity (HET) in both the workload and the distributed infrastructure has repeatedly been proposed for improving static scheduling of scientific workflows on computational grids (Topcuoglu et al., 2002; Yu and Buyya, 2006). In ERA, exploitation of heterogeneity is realized as follows.

When, at the current time t_c, a machine j is able to execute a new task, ERA obtains task runtime estimates $\hat{e}_{t_c}^{i,j}$ for all bags of tasks B_i containing at least one ready task. For all these bags of tasks, it determines the estimated runtime $\hat{e}_{t_c}^{i,min}$ if a task of that bag were to be scheduled on the most well-performing machine other than j. Similar to the Sufferage heuristic introduced in Section 2.2.4.2, ERA then computes a sufferage score for all bags of tasks as the difference between $\hat{e}_{t_c}^{i,j}$ and $\hat{e}_{t_c}^{i,min}$. The sufferage score for a bag of tasks B_i denotes by how much the expected execution duration of a task of that bag would suffer if that task were assigned to machine j as opposed to its most suitable machine (other than j). ERA then selects a ready task from the bag with the lowest sufferage score for execution on j. In doing so, ERA assigns tasks to machines based on suitability, i. e., based on how quickly these tasks are expected to be executed by their assigned machines as opposed to other machines.

Notably, tasks can vary substantially in their execution duration. Therefore, ERA normalizes runtime estimates across all machines by division through their average. If this normalization were to be omitted, the sufferage scores for bags of computationally intensive tasks with a generally high execution duration would dominate the sufferage scores of bags of smaller tasks. Due to their larger absolute sufferage value (on average), those larger tasks would be selected less frequent in general. Consider the following example and its illustration in Figure 4.5.

Example *Assume that* ERA *schedules a workflow comprising two bags of tasks B_v and B_w on three machines x, y, and z. By modeling tasks of both bags as two separate Wiener processes,* ERA *determines runtime estimates $\hat{e}_{t_c}^{v,x} = 6$, $\hat{e}_{t_c}^{v,y} = 10$, $\hat{e}_{t_c}^{v,z} = 14$, $\hat{e}_{t_c}^{w,x} = 90$, $\hat{e}_{t_c}^{w,y} = 95$, and $\hat{e}_{t_c}^{w,z} = 115$. Evidently, tasks of bag B_v are more suitable to be assigned to machine x than tasks of bag B_w. To compare estimates between bags of tasks and derive scheduling decisions based on sufferage values, these estimates are first normalized to $\hat{e}_{t_c}^{v,x} = 0.6$, $\hat{e}_{t_c}^{v,y} = 1$, $\hat{e}_{t_c}^{v,z} = 1.4$, $\hat{e}_{t_c}^{w,x} = 0.9$, $\hat{e}_{t_c}^{w,y} = 0.95$, and $\hat{e}_{t_c}^{w,z} = 1.15$. When machine y is ready to execute a new task,* ERA *computes sufferage values $s_v = \hat{e}_{t_c}^{v,y} - \hat{e}_{t_c}^{v,min} = \hat{e}_{t_c}^{v,y} - \hat{e}_{t_c}^{v,x} = 0.4$ and $s_w = \hat{e}_{t_c}^{w,y} - \hat{e}_{t_c}^{w,min} = \hat{e}_{t_c}^{w,y} - \hat{e}_{t_c}^{w,x} = 0.05$ for B_v and B_w, respectively. Due to its lower sufferage value,* ERA *selects a task from B_w for execution on y, leaving tasks from B_v for machine x, which is more suitable for execution of these tasks.*

4.2.4 Runtime Complexity

Updating a Wiener Process Model $\widehat{W}^{i,j}$ upon termination of a task and determination of its elapsed runtime can be realized in runtime $\mathcal{O}(1)$: To update $\widehat{W}^{i,j}$, we only adjust the average increment $\overline{d}^{i,j}$ as well as the sampling standard deviation $s^{i,j}$ for the bag-of-tasks-machine-pair (B_i, j) (Chan and Lewis, 1979). To keep models updated throughout

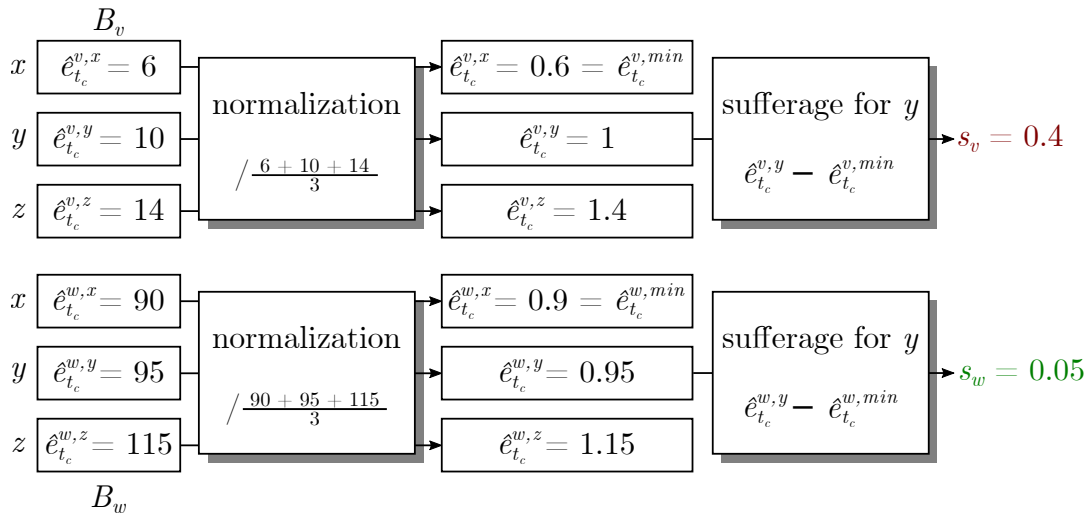


Figure 4.5: An example that illustrates how the exploitation heuristic determines a bag of tasks to select a task from. Runtime estimates $\hat{e}^{i,j}_{t_c}$ are provided for bags of tasks B_i with $i \in \{v, w\}$ and machines $j \in \{x, y, z\}$. When machine y is ready to execute a new tasks, all estimates are first normalized. Subsequently, sufferage scores are computed for each bag of tasks. Finally, a task is selected from the bag of tasks with the lowest sufferage score, i. e., the smallest increase over its minimum normalized runtime estimate across all machines. In this example, this is bag B_w due to its low sufferage score of 0.05.

execution of the whole workflow, a total runtime in $\mathcal{O}(n)$ is required, where n is the number of tasks composing the workflow.

The runtime complexity of ERA is in $\mathcal{O}(b \cdot m)$ for a single scheduling decision, where b is the amount of bags of tasks and m is the number of available machines: To select a task for execution on machine j, runtime estimates for up to b bags of tasks on all m machines are considered (see Algorithm 1). Note that the number n of tasks composing the scientific workflows provides an upper bound for b. To schedule all tasks of a workflow, the runtime complexity is in $\mathcal{O}(n \cdot b \cdot m)$, since ERA is invoked n times.

4.3 Evaluation

In this section, we empirically examine the exploitation, replication, and adaptation heuristics provided by ERA. In particular, we analyze (i) how these heuristics can be configured and (ii) how they cope with computational infrastructures subject to varying levels of variability. In addition, we compare the performance of ERA against other workflow schedulers. Experiments throughout this section have been employed using the DynamicCloudSim simulation toolkit introduced in Chapter 3. An evaluation experiment of ERA on real cloud infrastructure is presented in Section 5.2.3 on page 106.

Input: bag-of-tasks workflow $(S, \mathcal{B})$; machine j that is able to execute an additional task; current time t_c; adaptivity parameter $0 < \alpha \leq 0.5$; replication parameter $\rho \geq 0$
Output: task to be executed on j or **null**, if no tasks are available

$\mathcal{B}_{ready} \coloneqq \{B \in \mathcal{B} \mid \exists t \in B : t \text{ is ready}\}$ // bags of ready tasks, (see Section 8)
$\mathcal{B}_{run} \coloneqq \{B \in \mathcal{B} \mid \exists t \in B : t \text{ is running}\}$ // bags of running tasks
$r \coloneqq$ number of currently running task replicates
// check if an already running task is to be replicated
$\mathcal{B}_{select} \coloneqq \mathcal{B}_{ready}$ // bags of tasks to choose from
if $\mathcal{B}_{select} = \varnothing$ **then** // all ready tasks running
  if $r \geq \rho$ **then** // maximum number of allowed replicas reached
    return null
  end
  $\mathcal{B}_{select} \coloneqq \mathcal{B}_{run}, \quad \alpha \coloneqq 0.5$ // disable adaptivity if task is to be replicated
end
// determine bag of tasks with lowest sufferage for machine j
$M \coloneqq$ set of machines in the distributed infrastructure
$B_{min} \coloneqq$ **null** // bag of task with minimum sufferage
$s_{min} \coloneqq \infty$ // minimum sufferage value
foreach $B_i \in \mathcal{B}_{select}$ **do**
  $\hat{e}_{t_c}^{i,j} \coloneqq \alpha$-quantile of distribution of $\widehat{W}_{t_c}^{i,j}$ // runtime est. for tasks from B_i on j
  $\hat{e}_{t_c}^{i,min} \coloneqq \infty$ // minimum runtime estimate for tasks from B_i on any machine
  $\hat{e}_{t_c}^{i,sum} \coloneqq \hat{e}_{t_c}^{i,j}, \quad \hat{e}_{t_c}^{i,num} \coloneqq 1$ // variables for normalizing sufferage values
  foreach $k \in M \setminus j$ **do**
    $\hat{e}_{t_c}^{i,k} \coloneqq \alpha$-quantile of the distribution of $\widehat{W}_{t_c}^{i,k}$
    if $\hat{e}_{t_c}^{i,k} < \hat{e}_{t_c}^{i,min}$ **then**
      $\hat{e}_{t_c}^{i,min} \coloneqq \hat{e}_{t_c}^{i,k}$
    end
    $\hat{e}_{t_c}^{i,sum} \coloneqq \hat{e}_{t_c}^{i,sum} + \hat{e}_{t_c}^{i,k}, \quad \hat{e}_{t_c}^{i,num} \coloneqq \hat{e}_{t_c}^{i,num} + 1$
  end
  $\hat{e}_{t_c}^{i,avg} \coloneqq \frac{\hat{e}_{t_c}^{i,sum}}{\hat{e}_{t_c}^{i,num}}, \quad s_i \coloneqq \frac{\hat{e}_{t_c}^{i,j} - \hat{e}_{t_c}^{i,min}}{\hat{e}_{t_c}^{i,avg}}$ // normalized sufferage when scheduling i on j
  if $s_i < s_{min}$ **then**
    $s_{min} \coloneqq s_i$
    $B_{min} \coloneqq B_i$
  end
end
return ready task from B_{min}

Algorithm 1: The ERA algorithm for the adaptive scheduling of scientific workflows. We assume that sufficient runtime measurements have already been gathered such that Wiener process models for all tasks on all machines are available.

We introduce a synthetic evaluation workflow that maximizes heterogeneity between tasks in Section 4.3.1. In Section 4.3.2, we outline how we expect the three heuristics provided by ERA to cope with the aspects of variability modeled by DynamicCloudSim and introduced in Sections 3.3.2 to 3.3.4. In Section 4.3.3, we examine ERA's ability to exploit heterogeneity (HET) in the underlying computational infrastructure. We analyze ERA's ability to adapt to dynamic performance changes at runtime (DCR) and discuss how to determine suitable settings for the α parameter of ERA's adaptivity heuristic in Section 4.3.4. In Section 4.3.5, we analyze how ERA's replication heuristic increases robustness to straggler machines and failures during task execution (SAF). Finally, in Section 4.3.6 we evaluate the performance of ERA with different configurations against the workflow schedulers previously evaluated and discussed in Section 3.6.

4.3.1 Synthetic Evaluation Workflow

To examine the interplay of ERA heuristic in an ideal environment that complies with items 1 and 3 of the conditions listed in Section 4.2, we designed a synthetic test workflow that maximizes heterogeneity between tasks. It comprises three bags of tasks, one each for I/O-, CPU-, and network-intensive tasks. The workflow is based on typical extract, transform, load (ETL) processes, which are common in database and data warehouses applications. Specifically, it emulates a data-parallel application that first reads substantial amounts of input data from local storage, then performs CPU-intensive transformations on this data, and finally loads the results of computation, which are smaller in size than raw input data, to external storage.

There are data dependencies between I/O- and CPU-bound tasks, as well as between CPU- and network-bound tasks (see Figure 4.6 for an abstract illustration). Each bag contains 768 tasks and each task is expected to take five minutes on a machine of type m1.small (one compute unit, 1.7 GB of main memory) with all of DynamicCloudSim's variability parameters HET, DCR, and SAF set to zero. When executed on a set of eight such homogeneous and stable virtual machines, we expect simulated workflow execution to take approximately one day ($3 \cdot 768 \cdot 5 \div 8 = 1440$ minutes).

4.3.2 Expected Performance of ERA's Heuristics

The exploitation heuristic presented in Section 4.2.3 is integral to ERA and, thus, cannot be altered or omitted. It ensures that tasks are assigned to machines which, compared to other machines, have performed well when executing such tasks in the past. We expect a configuration of ERA limited to this heuristic to outperform knowledge-free schedulers like FCFS (see Section 2.2.4.1) on heterogeneous infrastructures.

ERA's adaptation heuristic, as outlined in Section 4.2.1, assures that the collection of runtime estimates for all (bags of) tasks on all machines stays up-to-date and complete. Through its α parameter, it balances the accuracy of runtime estimation against the necessity to occasionally re-explore task-machine-assignments previously observed to be unfavorable. We expect the adaptation heuristic to increase robustness to environments subject to dynamic performance changes at runtime (DCR). However, this increased

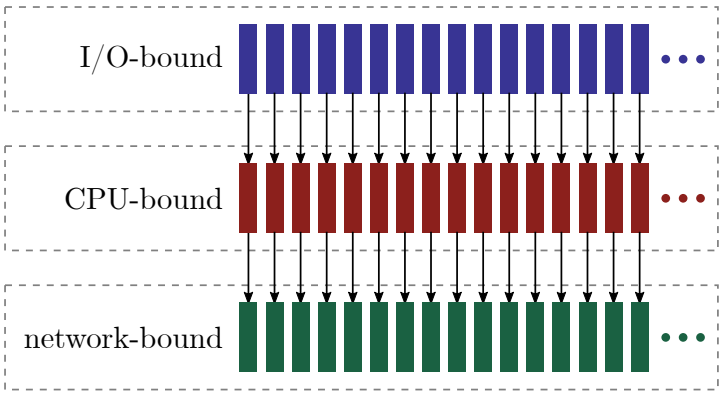


Figure 4.6: An illustration of the synthetic evaluation workflow. The workflow comprises three bags of tasks (I/O-, CPU-, and network-bound). Due to space constraints, the displayed instance of the workflow only contains 16 (as opposed to 768) tasks per bag. Each vertical block represents one task.

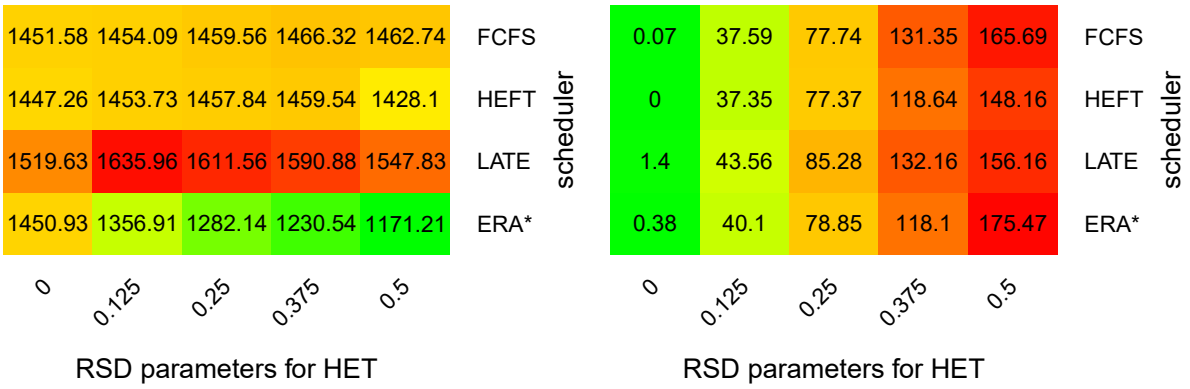
robustness should come at the cost of a degradation in observed performance for higher values of α.

Finally, the replication heuristic introduced in Section 4.2.2 ensures that machines with a (currently) high performance are not idly waiting for potentially critical tasks to terminate on slower machines. To this end, it provides a ρ parameter through which the amount of tasks to be speculatively replicated can be controlled. We anticipate this replication heuristic to improve performance in the presence of straggler machines and failures during task execution (SAF).

4.3.3 Exploiting Heterogeneity

In this experiment, we evaluated ERA's ability to exploit heterogeneity both in the performance requirements of a scientific workflow's tasks and in the resources provided by the computational infrastructure. To this end, we simulated the execution of the synthetic evaluation workflow described in Section 4.3.1 for workflow schedulers FCFS, HEFT, LATE, and ERA with disabled adaptivity and replication heuristics. Similar to the experiments outlined in Section 3.6.1, we steadily increased the heterogeneity (HET) introduced by DynamicCloudSim through its heterogeneity RSD parameters. Other aspects of variability introduced by DynamicCloudSim (DCR and SAF) were disabled, resulting in an increasingly heterogeneous, yet stable (simulated) computational infrastructure. For each combination of scheduler and heterogeneity setting, we simulated workflow execution 1000 times on eight virtual machines. Observed values for median and standard deviation are shown in form of a heatmap graphic in Figure 4.7. Mean values are listed in Table A in the Appendix.

As expected, employing knowledge-free FCFS scheduling resulted in simulated workflow execution times close to the expected makespan of one day (1440 minutes). Similar observations were made for HEFT, which comes as a surprise considering that HEFT has been developed to cope with heterogeneity. HEFT assigns tasks to machines based on



(a) median runtimes in minutes

(b) standard deviation of runtimes

Figure 4.7: Increasing levels of heterogeneity in the computational infrastructure (HET) and their effects on workflow runtime using different schedulers. Of the examined schedulers, only ERA is able to exploit heterogeneity and improve median workflow runtime below 1440 minutes. *ERA with only its exploitation heuristic enabled and replication and adaptation heuristics disabled.

their upwards rank score, which is mostly influenced by the amount and computational cost of upstream tasks (see Section 2.2.4.2). It therefore exploits heterogeneity in a workflow's structure, giving a higher priority to critical tasks blocking the execution of large numbers of upstream tasks (pipeline blockers). However, the synthetic evaluation workflow has a simple workflow structure with many similar tasks at identical depths within the workflow. Tasks belonging to the same bag will therefore have a similar upwards rank score, which undermines HEFT's scheduling strategy.

The absence of pipeline blockers in the evaluation workflows also explains LATE's subpar performance: If all tasks are comparable in their computational cost, LATE has little to gain from replication. By reserving 10 % of available resources for replication however, LATE effectively wastes resources, which results in simulated workflow makespans below FCFS. In contrast, ERA's exploitation heuristic is able to determine favorable task-machine-assignments in the face of heterogeneity, improving simulated workflow runtime well below the baseline of 1440 minutes.

All of the examined schedulers have in common that increasing the heterogeneity in the computational infrastructure leads to higher variations in simulated workflow runtime. The reason for this is that a highly heterogeneous infrastructure may comprise individual compute nodes with outstandingly high or low performance values, resulting in some outliers among simulated workflow runtimes.

4.3.4 Adapting to Dynamic Changes at Runtime

The α parameter of ERA's adaptation heuristic balances the conservatism of assigning tasks to machines that have been tried and tested to work well for similar tasks (i. e., tasks belonging to the same bag) against the curiosity of occasionally re-evaluating

assignments to other machines. Lower values for α (with $0 < \alpha \leq 0.5$) should be reflected in a faster degradation of runtime estimates over time, since runtime estimates are based on the α-quantile of random variable $\widehat{W}_{t_c}^{i,j}$.

4.3.4.1 Adjusting the α parameter

To evaluate the interaction between α and dynamic changes in the infrastructure (DCR), we simulated the execution of the synthetic test workflow described in Section 4.3.1 on eight virtual machines. Similar to the experiments in Section 3.6 on page 54, we (i) gradually increased DynamicCloudSim's persistent performance change RSD parameters, while (ii) setting RSD parameters for short-term fluctuations to a fixed value of 0.025 across all runs and (iii) disabling all other models of variability (HET and SAF) in DynamicCloudSim. However, there were two differences to the experiments in Section 3.6: First, we increased the RSD parameters for DCR all the way up to 1.25 to evaluate ERA's adaptivity heuristic for infrastructures subject to very high levels of variability. Secondly, note that DynamicCloudSim models performance variations by sampling from a normal distribution. Hence, for RSD parameters beyond 1.0, DynamicCloudSim samples from distributions with a higher standard deviation than the mean. For such high RSD values, we observed occasional spikes in virtual machine performance that allow for the rapid execution of a whole bag of tasks in a few minutes. In this experiment, we therefore capped the maximum reachable performance coefficient of virtual machines to 2. This way, virtual machine performance heavily fluctuates at RSD parameters 1.25, but fluctuations are restricted to a window between 0 and 2 times the average performance.

We disabled replication of tasks in ERA and repeatedly simulated workflow execution 1000 times for different combinations of α and DynamicCloudSim's DCR parameters. Figure 4.8 displays the results of the experiment as a heatmap of observed median values (see Tables B and C in the Appendix for mean and standard deviation values, respectively). It indicates a setting of $\alpha = 0.2$ to provide a good tradeoff between exploitation and adaptation goals. This is especially true for low to moderate levels of DCR, as observed in Amazon EC2 (consider Table 3.1 on page 43 for RSD values observed in EC2 and assumed by DynamicCloudSim). It also reveals that decreasing α beyond this point does not further increase robustness to dynamic performance changes at runtime. Evidently, setting α to values below 0.05 results in runtime estimates degrading too quickly for the scheduler to make use of them.

Finally, the figure also showcases that at a certain level of variation in performance (RSD parameters for DCR beyond 0.75), the effects of adaptivity diminish. Surprisingly, disabling the adaptivity heuristic altogether (i. e., setting α to 0.5) results in the best performance for such configurations. Apparently, if performance fluctuations are too high, the cost of repeatedly re-evaluating task runtime outgrows its potential.

Note that an optimal parameter setting for α depends not only on the variability of the computational infrastructure, but also on the frequency of task runtime measurements: Frequently executed workflows with large bags of short-running tasks may warrant lower values for α, while rarely executed, smaller workflows with long-running

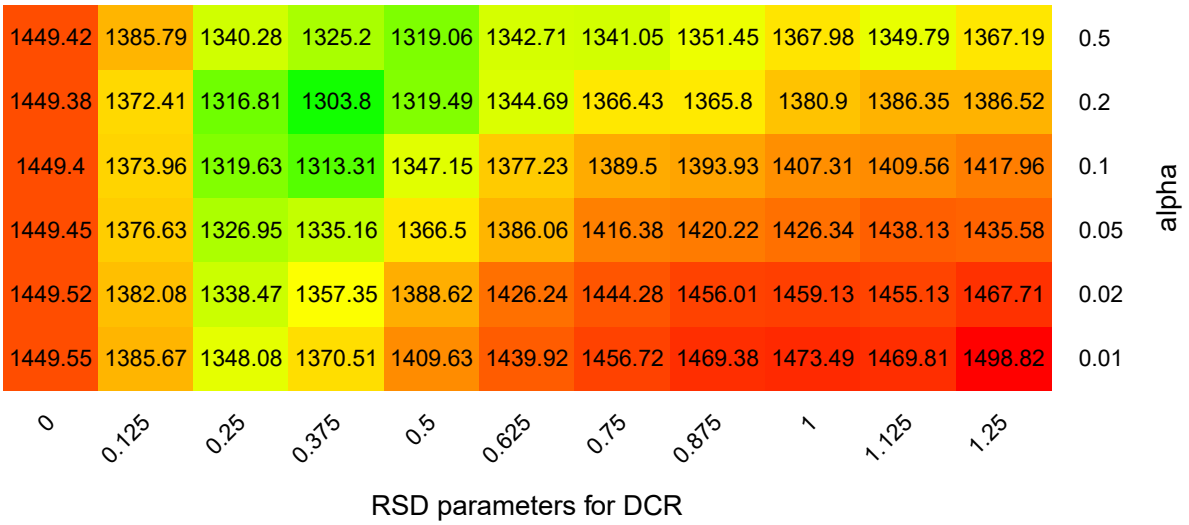


Figure 4.8: Median workflow makespan in minutes at increasing levels of dynamic performance changes at runtime in the computational infrastructure (DCR) and ERA's ability to adapt to these changes by means of its α parameter. Lower values for α translate into more frequent re-evaluations of bag-of-task-machine-assignments. $\alpha = 0.2$ seems to offer a good compromise between exploitation and adaptation.

tasks may require higher α values. See Figure 4.9 for an exemplary visual assessment of the adaptation heuristic with α set to 0.2.

4.3.4.2 Performance Gains through Adaptivity

Having determined a suitable setting for the α parameter, we compared the performance of ERA's exploitation and adaptation heuristics in the face of dynamic performance changes at runtime to the schedulers evaluated in Section 3.6. To this end, similar to the experiment in Section 3.6.2.2, we steadily increased RSD parameters for DCR from 0 to 0.5. DynamicCloudSim's other aspects of variability (HET and SAF) were disabled. We simulated the execution of the synthetic evaluation workflow 1000 times for each configuration of scheduler and DCR parameter setting. Observed medians and standard deviations in workflow runtime are shown in Figure 4.10 in form of a heatmap graphic. Mean values are listed in Table D in the Appendix.

Similar to the heterogeneity experiments described in Section 4.3.3, employing a knowledge-free FCFS scheduling policy resulted in workflow makespans around 1440 minutes. Similarly, we observed the performance of LATE to be worse than FCFS due to LATE reserving 10 % of available resources for replication. In contrast to the results from Figure 4.7 however, the performance of HEFT scheduling quickly diminishes when performance fluctuations at runtime are introduced. This finding confirms the previously discussed shortcomings of static schedulers like HEFT. Conversely, the performance of ERA's exploitation and adaptation heuristics actually improved with the amplitude of

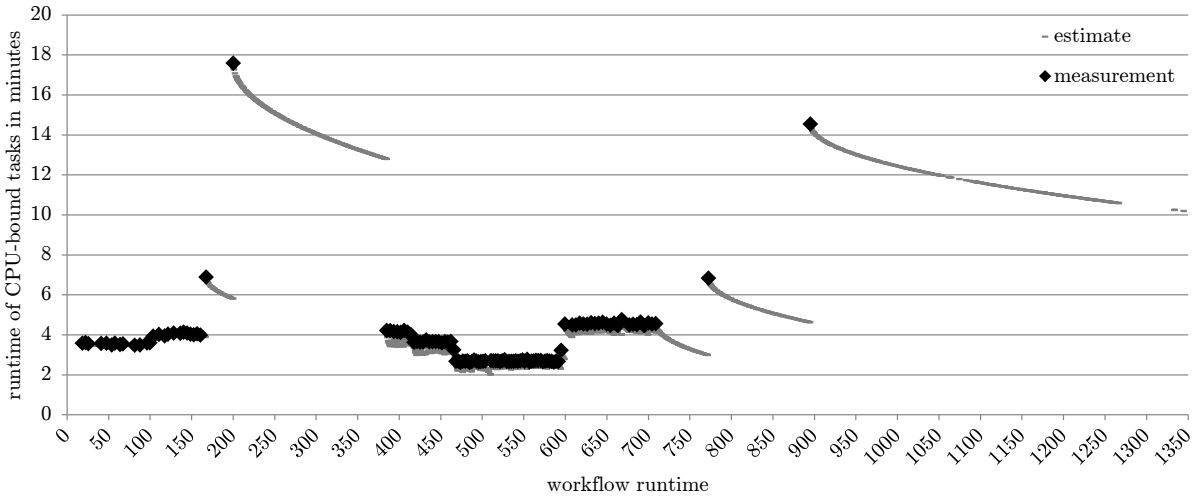
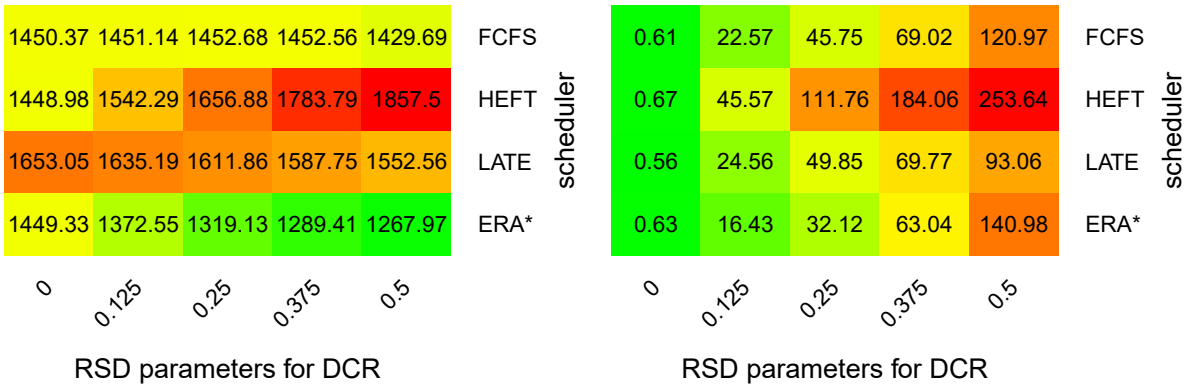


Figure 4.9: Development of measured task runtimes and determined runtime estimates for CPU-bound tasks on a single virtual machine. Parameters for DCR and α were set to 0.5 and 0.2, respectively. The plot illustrates the interaction between ERA's exploitation and adaptation heuristics. In intervals of favorable CPU performance (i. e., observed runtimes around or below the average of five minutes), ERA assigns mostly CPU-bound tasks to the machine. Upon measuring increased runtimes, ERA omits further assignments of CPU-bound tasks in favor of other tasks. However, runtime estimates slowly degrade such that, eventually, a CPU-bound task is assigned to the machine again.



(a) median runtimes in minutes

(b) standard deviation of runtimes

Figure 4.10: Increasing levels of dynamic performance changes at runtime in the computational infrastructure (DCR) and their effects on workflow runtime using different schedulers. While the performance of FCFS and LATE are stable, only ERA is able to exploit the heterogeneity introduced through performance changes. *ERA with α set to 0.2 and disabled replication heuristic.

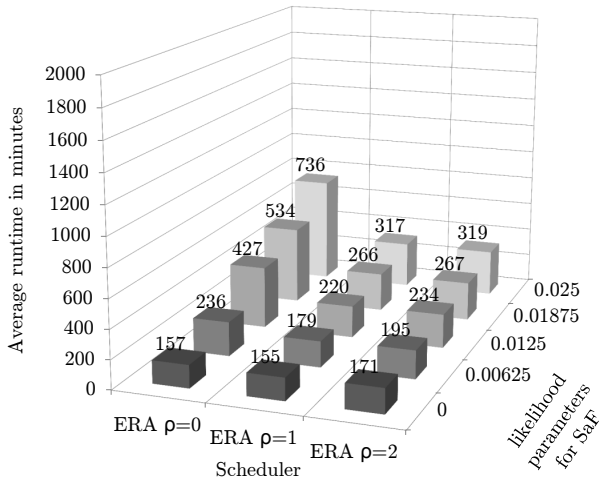


Figure 4.11: Effects of straggler virtual machines and failed tasks (SAF) on workflow runtime using ERA in DynamicCloudSim. Mean execution times of the SNV calling workflow described in Section 3.6.1 are reported. This plot extends Figure 3.9b on page 56. It showcases that the introduction of ERA's replication heuristic leads to severely increased robustness to SAF.

introduced performance changes at runtime. Despite the computational infrastructure starting out entirely homogeneous in this experiment, the (unstable) heterogeneity introduced by these performance fluctuations is adapted to and exploited by ERA.

Observed variance in performance was lowest for LATE however, which can be explained as follows: For high levels of DCR, individual virtual machines will occasionally be subject to heavily degraded performance for a time frame of several hours. This can prolong workflow makespan by several hours if one of the last remaining tasks is assigned to such a machine during workflow execution. Through its replication strategy, LATE provides unmatched robustness in such scenarios.

4.3.5 Replication of Straggling or Failing Tasks

In this experiment, we examined ERA's replication heuristic and its ability to increase robustness to straggler machines and failed task executions (SAF). As outlined previously and confirmed by LATE's subpar performance in Figures 4.7 and 4.10, the simple structure of the synthetic evaluation workflow is ill-suited for such an evaluation. Instead, we therefore extended the experiment outlined in Section 3.6, employing a variant calling workflow that contains several pipeline blockers.

We simulated the execution of this workflow on eight virtual machines with two compute units and 3.75 GB memory each. Simulations were repeated 100 times for SAF settings between 0 and 0.025 and different configurations of ERA's replication heuristic. The results of the experiment in the form of observed mean workflow runtimes extend Figure 3.9b on page 56 and are shown in Figure 4.11. The medians and standard deviations of workflow runtime are listed in Tables E and F in the Appendix.

We observed that ERA, through introduction of its replication heuristic, provides robustness to even high occurrences of stragglers and failures well beyond numbers encountered in Amazon EC2. While the degradation of measured workflow runtimes for high SAF values was still notably lower for LATE, note that LATE implements a substantially more aggressive replication strategy. In contrast to LATE, ERA only replicates tasks when resources are idle. Replication in ERA therefore comes at a much lower cost.

Notably, increasing the maximum number of concurrently running task replicates through ERA's ρ parameter did not result in increased robustness to SAF. However, higher numbers of ρ resulted in performance gains when the maximum amount of concurrent tasks accepted by a single machine (i. e., its task slots) was increased beyond one (results not shown). The reason for this finding is that if ρ is set to one, yet machines are configured to run two tasks in parallel, a replicate of a pipeline blocker running on a straggler machine can, in some instances, be assigned to the same straggler machine. Hence, we recommend setting ρ to the number of task slots per machine in the computational infrastructure.

4.3.6 Scheduling Performance

In this final experiment, we evaluated the performance of ERA against other schedulers on a (simulated) computational infrastructure resembling Amazon EC2. To this end, we simulated the execution of the SNV calling workflow on eight virtual machines corresponding to EC2 instances of type m1.medium (i. e., two compute units and 3.75 GB of memory each). Simulations were repeated 1000 times for each scheduler and DynamicCloudSim was configured with its default parameters (see Table 3.1 on page 43). The results of this experiment are shown in Figure 4.12 as well as in Table G in the Appendix.

We observed comparable mean workflow execution times across schedulers FCFS, HEFT, LATE, and ERA with disabled replication heuristic ($\alpha = 0.2$, $\rho = 0$). Both LATE and ERA with disabled replication significantly outperformed the baseline of FCFS scheduling (two-tailed t-tests with p-values of 0.009 and 0.032, respectively). Both schedulers, as well as HEFT, also featured a substantially lower variation in observed runtime. However, while LATE achieves these gains in performance and robustness by means of aggressive task replication, ERA achieves them by exploiting heterogeneity and adapting to changes. Enabling ERA's less invasive replication strategy ($\rho = 1$) increases performance further, leading to significantly improved workflow runtimes over LATE (p-values of $5.86 \cdot 10^{-30}$ and $8.43 \cdot 10^{-37}$ for ER and ERA, respectively). In addition, ERA's replication heuristic also strongly reduces the variation in observed makespans, in particular when coupled with the adaptation heuristic ($\alpha = 0.2$).

In closing, we find that ERA provides robustness to all aspects of variability typically encountered on shared and distributed computational infrastructures (HET, DCR, and SAF). Reductions in the mean values, medians, and standard deviations of workflow runtimes below values reached by established schedulers are achieved via three heuristics. While each of these heuristics contributes to the overall scheduling quality, neither of

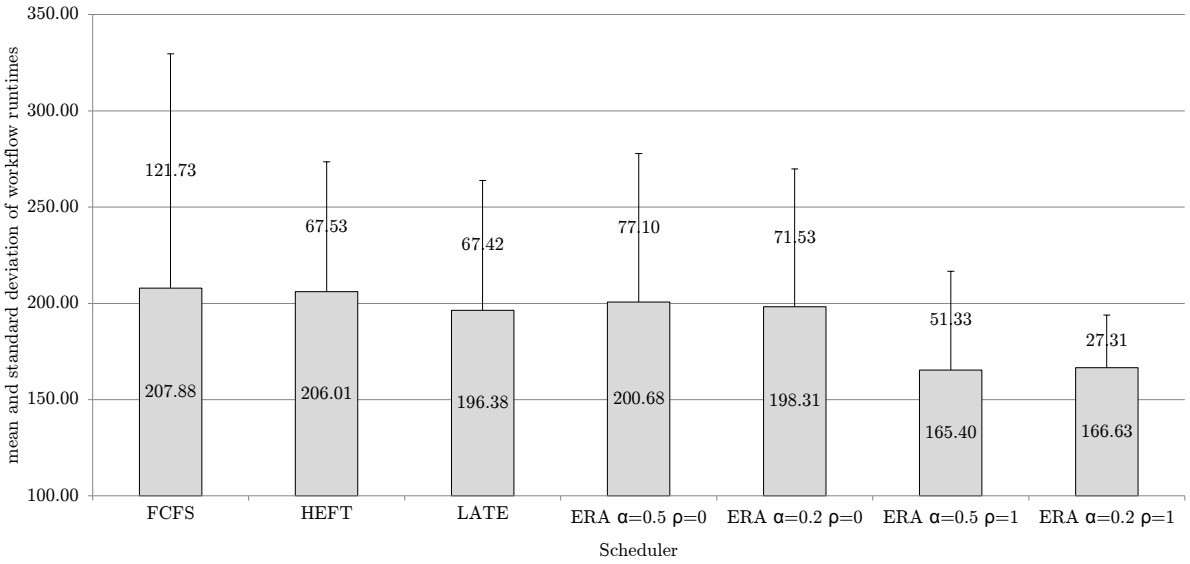


Figure 4.12: Mean values and standard deviations of simulated SNV calling workflow runtimes are shown for different schedulers.

them adds substantial computational overhead. We also observed that all three heuristics are at their best when employed in combination with one another.

Clearly, simulation can never perfectly model a physical computational environment. Also note that performance variations of machines in DynamicCloudSim are normally distributed by default and ERA assumes differences in runtime measurements to be normally distributed. Since performance variations over time need not always be normally distributed in practice (see Section 3.3.2), the evaluation in DynamicCloudSim could unintendedly favor ERA. A complementary evaluation of ERA's performance on actual cloud infrastructure is therefore presented in Section 5.2.3.

4.4 Related Work

In this section, we give an overview of the state-of-the art in task runtime estimation and adaptive scheduling of scientific workflows. While, as outlined in the introduction of this chapter, we find that these two concepts depend on one another and should be meshed into an integrated component, they have traditionally been researched separately. Consequently, we survey related approaches for both of these concepts separately in Sections 4.4.1 and 4.4.2, respectively.

4.4.1 Task Runtime Estimation

Task runtime estimation has been studied extensively over the last decades. Approaches can generally be subdivided into three classes. The most straightforward class of approaches involves determining the distribution of measured task runtimes and obtaining

a forecast of future task runtimes based on this distribution, e. g., via sampling or confidence intervals (Gibbons, 1997). The second group of approaches incorporate a set of quantifiable features (e. g., task runtime, size of input data, assigned compute node, task metadata) and employ machine learning methods such as classification, nearest neighbor search, artificial neural networks, or multi-dimensional regression analysis to determine runtime estimates (Iverson et al., 1999).

The third group of approaches consider historical task executions as time series, thus taking into account the temporal sequence at which observations were taken for inferring current estimates. A characteristic property shared by most time-series-based methods is that they acknowledge more recent measurements to likely be more reliable than older, possibly outdated measurements. This is usually achieved by either discarding old measurements entirely or by assigning higher weights to more recent measurements. Consequently, this group of prediction methods is able to detect, model, and adapt to dynamic performance changes at runtime, which are commonly encountered in distributed computational infrastructure shared between multiple users (see Sections 2.2.2.3 and 3.1).

Our approach of estimating task runtime by modeling it as a Wiener process is time-series-based: A Wiener process is most strongly influenced by the most recently observed runtime, yet also incorporates earlier measurements and the time at which they were observed. We therefore focus our account of related work on other time-series-based methods of performance prediction. This includes methods for task runtime estimation and system load prediction. Both fields are closely related to one another and methods can usually be applied interchangeably. See Table 4.1 for a comparative overview of both the commonalities and differences between related methods and ERA.

4.4.1.1 The Pioneers of Time-series-based Performance Prediction

To the best of our knowledge, Devarakonda and Iyer (1989) were the first to publish a method of time-series-based performance prediction. Observed tasks are represented as points in a space comprising the dimensions of runtime, peak memory usage, and file I/O. Using the k-means algorithm, clusters of tasks are determined, such that, for instance, one cluster comprises memory-intensive tasks, whereas another is composed of CPU-intensive tasks. Subsequently, for any given bag of tasks, a Markov model is assembled which, based on the order of execution and cluster membership of past tasks belonging to the same bag, models the likelihood for subsequent tasks of that bag to change cluster membership. This model is used to forecast the characteristics (in the form of the aforementioned three dimensions) of a to-be-executed task based on the latest observed task belonging to the same bag along with its cluster transition likelihoods.

In contrast to ERA, this model is difficult to improve on-the-fly when new runtime measurements become available during workflow execution. In addition, it also does not provide any solutions for determining the ideal number of clusters for the k-means algorithm. Instead, the number of clusters is simply set to seven by default.

Arguably the most influential approach of time-series-based performance prediction is the Network Weather Service (NWS) developed by Wolski et al. (1999) The NWS is

a distributed performance forecasting framework that continuously measures CPU and network performance, based on which it computes current load forecasts. It employs a comprehensive collection of different prediction methods (Wolski, 1998), which include: (i) the sliding window average (SW) based on the last p measurements, (ii) an autoregressive estimator (AR), i. e., a weighted average to which more recent measurements contribute stronger than older measurements; (iii) a predictor based on exponential smoothing (ES) that determines the current runtime estimate based on the latest measurement and the last runtime estimate prior to this measurement; (iv) basic prediction methods like the last measurement or the arithmetic mean, the median, or any other quantile of the distribution of measurements.

These methods are employed simultaneously, and, to optimize prediction accuracy, the method which has exhibited the smallest prediction error (measured as the deviation between previously predicted and observed performance) is reported as the current forecast. To keep intrusiveness of the prediction system low, the frequency at which new forecasts are generated is adaptively adjusted based on the accuracy of earlier forecasts. The NWS is able to consistently forecast CPU availability up to five minutes in advance with an error of around 10 % (Wolski et al., 2000). However, Sonmez et al. (2009) found the estimators included in the NWS to perform subpar on shared computational resources subject to bursts in resource contention.

In contrast to ERA, the ensemble of methods employed by the NWS do not model the variance between runtime measurements, but only focus on some form of weighted average.

4.4.1.2 Extensions and Alternatives to the NWS

Many subsequently presented time-series-based approaches are comparable to the NWS in that they employ ensembles and variations of SW, AR, ES, and basic estimators. Gao et al. (2005) proposed to degrade the weight of the latest measurement in ES over time. The determined runtime estimates are employed by a sampling-based scheduler, which leaves room for (re-)exploring node performance, while also exploiting favorable task-machine assignments. Dobber et al. (2007) compare different adaptive exponential smoothing (AES) techniques which continuously adjust weights based on the observed forecast error. This way, the weight of the latest measurement is increased if, for instance, the last forecast was found to be inaccurate. Tsafrir et al. (2007) proposed the average of the last two measurements as an additional estimator. Wu et al. (2010) suggested to apply Kalman and Savitzky-Golay filters to measurement data to reduce noise prior to applying an AR estimator. Herbst et al. (2014) proposed an extensive ensemble of predictors for forecasting system load, including more computationally intensive predictors such as the autoregressive integrated moving average (ARIMA) model. Predictors are categorized according to their computational complexity and selection of feasible predictors is guided at runtime by a decision tree based on, for instance, the amount of available measurements as well as the computational cost and observed accuracy of employed predictors.

Similar to the NWS and as shown in Table 4.1, the disadvantages of all these systems when compared to ERA are that they do not have notions of time or uncertainty (see Table 4.1): While they respect the temporal sequence of measurements, they do not model or incorporate how much time has passed between measurements and, in particular, since the last measurement. In addition, they often have been developed in isolation of the scheduling applications they are intended for or do not ensure that the scheduler maps tasks to resources such that a wide range of measurements is kept up-to-date.

As outlined earlier, task runtime estimation and system load prediction are closely related to one another, since approaches proposed for the former can be employed for the latter, and *vice versa*. However, it is not straightforward to directly translate predicted load performance into a task runtime estimate. An approach that attempts to perform this translation is the Running Time Advisor (RTA) proposed by Dinda (2002b). The RTA predicts task runtime in the form of confidence intervals based on (i) a forecast of system load predicted by an AR estimator and (ii) the nominal task runtime, i.e., task runtime if the machine were idle. Runtime estimates reported by the RTA have been employed in the real-time scheduling advisor (RTSA) (Dinda, 2002a), which determines suitable hosts for executing CPU-bound tasks in a distributed system of homogeneous hosts. In contrast to ERA, the major disadvantages of this technique are that, in practice, nominal task runtimes are often not available, tasks are not exclusively CPU-bound (but may be I/O-bound, network-bound, or memory-bound instead), and the computational infrastructure is oftentimes heterogeneous.

To obtain reliable estimates of queue waiting times for jobs in batch scheduling systems, Brevik et al. (2006) proposed the Binomial Method Batch Predictor (BMBP). It provides a conservative prediction method that estimates an upper bound of the .95 quantile of queue waiting times, disregarding the temporal order of measurements. We consider BMBP to be a time-series-based predictor, since it detects change points in observed performance. To this end, it tracks so-called "rare events", i. e., measurements beyond the .95 quantile (of measurements). Upon detection of a certain determinable number of successive rare events (e.g., three), the BMBP assumes the occurrence of a change point. As new measurements become available, the BMBP then phases out measurements taken prior to the suspected change point. Conversely, while ERA does not explicitly determine change points, it will, upon occurrence of a change point, notice an increased variance in runtime measurements. By means of its adaptation heuristic (see Section 4.2.1), ERA is then increasingly likely to schedule additional tasks of the same bag on that machine until measured runtimes are stable and variance decreases again.

Yang et al. (2003a) proposed several prediction strategies that forecast CPU load based on the current tendency, i. e., the difference between the latest and second-to-latest measurements $e_{t_n} - e_{t_{n-1}}$. Tendency-based strategies forecast an increase of the current runtime estimate $\hat{e}_{t_c}$ over e_{t_n} if the current tendency is positive (i. e., increasing) and e_{t_n} is not already too high above the mean (which might indicate an imminent "turning point"). The predicted increase of $\hat{e}_{t_c}$ can differ from e_{t_n} either by a fixed amount (static strategies), by an amount proportional to the current tendency (dynamic strategies), or by an amount relative to e_{t_n} (relative strategies). Yang et al. (2003b) extended the tendency-based, dynamic strategy to provide not only predictions for the load at a future

point in time, but also for the average of and variation in load for future time intervals. Based on this technique, they also proposed and evaluated a conservative scheduler that determines the expected interval length of a task and assigns it to a machine with a low predicted load average and variation in that interval.

Based on the work of Yang et al., several variations to the means of detecting and utilizing turning points in performance predictions have been proposed: Zhang et al. (2008) predict turning points in CPU load by comparing the current tendency against patterns that occurred in the past and that were associated with a turning point. Liu et al. (2011) predict task runtime by (i) detecting turning points and (ii) matching recent task runtime measurements against historical patterns found in between such turning points. All of these approaches assume variations in performance to follow tendencies (until a turning point occurs). In contrast to these tendency-based approaches, ERA rejects the concepts of tendencies and turning points. Instead, by modeling performance as a Wiener Process, fluctuations are assumed to occur entirely random, following no apparent pattern or tendency. Consequently, at any given time the likelihood of performance to decrease is just as high as it is to decrease.

4.4.2 Adaptive Scheduling

Mapping scientific workflows (or other jobs comprising heterogeneous tasks) onto distributed infrastructures is traditionally performed by knowledge-free or static schedulers (Braun et al., 2001; Yu et al., 2008). Some of these static schedulers have incorporated the standard deviations of task runtime measurements into scheduling decisions (e. g., Kamthe and Lee, 2011; Tang et al., 2011). To apply established (static) scheduling heuristics to computational infrastructures that are subject to instability and variability (as described in Sections 2.2.2.3 and 3.1), an obvious strategy is to (i) update runtime estimates during workflow execution and (ii) continuously re-evaluate previously determined scheduling decisions.

Prodan and Fahringer (2005) presented a re-scheduling heuristic that uses a genetic algorithm to determine a complete schedule of scientific workflow execution whenever a task finishes execution. In addition, it provides a task migration strategy comparable to the replication heuristic of ERA. This task migration component determines tasks progressing slower than expected based on (i) the current execution duration of that task or, if available, its progress rate and (ii) the makespans of earlier tasks belonging to the same bag. It then cancels potential straggler tasks and restarts their execution on another compute node. In contrast to this task migration strategy, ERA starts a replicate of the task on another machine, allowing the original instance of the task to continue for as long as the replicate does not terminate. This has the benefit of not prematurely canceling a task potentially close to completion. In addition, ERA only employs this strategy if idle resources are available, thereby only adding a very minor overhead.

Yu and Shi (2007) proposed adaptive HEFT (AHEFT) as an adaptive re-scheduling scheme based on the HEFT heuristic. Here, the HEFT scheduler (see Section 2.2.4.2) is employed anew for any remaining (unfinished) tasks as a consequence of any event of interest. Events of interest can, for instance, include the emergence of updated runtime

Table 4.1: Comparison of approaches towards time-series-based performance prediction. R: notion of recency, i. e., does the method weigh more recent measurements higher than older, possibly outdated measurements? T: notion of time, i. e., does the method model the amount of time that has passed between measurements or does it merely model the temporal sequence of measurements? U: notion of uncertainty, i. e., does the method include a measure of uncertainty in their prediction that increases with the amount of time that has passed since the last measurement? S: bundled with scheduler, i. e., has the prediction method been developed in conjunction with a scheduler?

references	prediction	methods	R	T	U	S
ERA, Section 4.1	task runtime	model task runtime as Brownian Motion / Wiener Process Model	✓	✓	✓	✓
Devarakonda and Iyer (1989)	task runtime	cluster invocations to obtain states of a Markov chain; utilize state transition likelihoods and latest observed states to obtain forecast	×	×	×	×
Wolski (1998); Wolski et al. (1999, 2000)	system load	combine SW, AR, ES, and naive estimators	✓	×	×	×
Gao et al. (2005)	task runtime	degrade smoothing factor of ES over time	✓	×	×	✓
Tsafrir et al. (2007)	task runtime	average of last two measurements	✓	×	×	×
Dobber et al. (2007)	task runtime	adapt smoothing factor of ES to observed error	✓	×	×	×
Wu et al. (2010)	system load	apply filters to reduce noise in measurements fed to AR estimation	✓	×	×	×
Herbst et al. (2014)	system load	select predictors from ensemble based on their computational complexity and available training data	✓	×	×	×
Dinda (2002b,a)	task runtime	derive task runtime from system load estimate and nominal task runtime	✓	×	×	✓
Brevik et al. (2006)	queue waiting time	discard outdated measurements upon detection of a change point	✓	×	×	✓
Yang et al. (2003a,b)	CPU load	prediction based on current observed tendency	✓	×	×	✓
Zhang et al. (2008)	CPU load	polynomial curve fitting (second or third order); pattern matching for turning point prediction	✓	×	×	×
Liu et al. (2011)	task runtime	detect turning points and match patterns in task runtime between turning points	✓	×	×	×

estimates for any task (e. g., due to the recent completion of another task), the discovery of new resources, or the unavailability of a previously available resource.

Similarly to AHEFT, Lee et al. (2009) implemented an adaptive scheduling mechanism for Pegasus. Job queues on execution sites are monitored and observed runtimes are compared against expectations. In case of sustained discrepancies, Pegasus computes a new schedule (for instance by employing HEFT scheduling) for workflow execution from the current point onwards. If the newly determined schedule promises favorable (estimated) execution times, it replaces the current schedule.

Both these approaches are comparable to the exploitation heuristic of ERA, since they optimize workflow makespan by determining favorable assignments of tasks to machines. However, ERA not only optimizes for the exploitation of heterogeneity, but also provides a replication strategy for stragglers as well as an adaptation strategy for keeping runtime estimates up-to-date. In contrast to ERA, approaches based on re-computing static schedules usually incorporate the workflow structure into their scheduling decisions, which comes at the cost of an increased runtime complexity: Most established static schedulers (such as HEFT) require a runtime in $\mathcal{O}(n^2 \cdot m)$ (Prodan and Fahringer, 2005), where n is the number of tasks and m is the number of machines (see Definition 7 on page 59). This runtime complexity can be problematic if workflows are large and schedules are re-computed repeatedly (up to n times). Another shortcoming of static schedulers is their inability to cope with iterative workflows (i. e., workflows containing recursive or conditional structures; see next chapter), in which new tasks are discovered only during workflow execution.

Dynamic just-in-time schedulers that limit their scheduling horizon to tasks ready for immediate execution can circumvent these problems. However, the performance of just-in-time schedulers, as for instance evaluated by Ramírez-Alcaraz et al. (2011), can be worse than that of static schedulers, since scheduling a task as soon as it is ready is often overly greedy and myopic (Malawski et al., 2015). The solution implemented in ERA is to (i) alleviate the effects of unfavorable task-machine-assignments by speculatively replicating potentially critical tasks when sufficient resources are available and (ii) delay scheduling decisions until resources are actually available and gather additional knowledge about tasks becoming ready for execution in the meantime.

Cai et al. (2017) implemented a scheduling strategy following the same rationale of delaying task scheduling in favor of gathering additional (global) workflow knowledge. The proposed algorithm has a different optimization goal than ERA: Here, scheduling is driven by the goal of minimizing virtual machine allocation cost in an elastic infrastructure-as-a-service cloud while guaranteeing termination within a given deadline. The scheduler operates by keeping available virtual machines busy and delaying execution of overflow tasks until allocation of an additional virtual machine becomes worthwhile. When a given threshold is reached, it tailors the number and specifications of additional virtual machines to the delayed tasks currently awaiting execution. To terminate workflow execution within the deadline constraint, the scheduler pessimistically estimates the execution duration of scheduled tasks as the sum of their runtime estimates and standard deviations.

Despite its different optimization goal, the algorithm implemented by Cai et al. is, to the best of our knowledge, the most closely related scheduler to ERA. We are not aware of any other just-in-time workflow scheduler that delays scheduling decisions until resources become available. Furthermore, we are not aware of even a single scheduling heuristic (besides ERA) that concerns itself with determining the accurate, up-to-date, and complete runtime estimates it requires to operate.

4.5 Summary

We presented ERA, an adaptive scheduler for running bag-of-tasks workflows on distributed infrastructures subject to instability, variability, and failure. ERA models task runtime as a Brownian motion, i. e., similar to the movement of particles in a fluid. During scientific workflow scheduling, ERA balances three heuristics against one another: (i) the exploitation heuristic determines task-machine-assignments that promise comparably low runtimes, (ii) the replication heuristic replicates bottleneck tasks when resources are idle, and (iii) the adaptation heuristic keeps runtime estimates up-to-date by occasionally re-evaluating task-machine-assignments with outdated measurements.

We presented and discussed results of a comprehensive evaluation of ERA using the simulation framework DynamicCloudSim. In this evaluation, we determined sensible parameter settings for ERA and compared performance against the schedulers previously discussed (and evaluated) in Section 3.6. ERA was implemented as part of the scientific workflow execution engine Hi-WAY, which is presented in the next chapter. An evaluation of ERA on real cloud infrastructure is also presented as part of the evaluation of Hi-WAY.

5 Executing Scientific Workflows on Hadoop

To deal with the ever-increasing amounts of data prevalent in today's science, scientific workflow management systems have to provide support for parallel and distributed storage and computation (Liu et al., 2015). In particular, as outlined in Section 2.2, they have to facilitate data-parallel workflow design and be able to adaptively schedule workflows on distributed computational infrastructure subject to various aspects of variability. However, while distributed resource managers like Hadoop YARN (Vavilapalli et al., 2013) or MESOS (Hindman et al., 2011) keep developing rapidly, established scientific workflow management systems, such as Taverna (Wolstencroft et al., 2013) or Pegasus (Deelman et al., 2015) are not able to keep pace (see Section 2.3).

A particular problem is that most scientific workflow management systems tightly couple their own custom workflow language to a specific batch scheduling system (as described in Section 2.2.3.1). Evidently, it can be difficult to configure and maintain such systems alongside other processing frameworks that are already present on the cluster, such as, potentially, modern distributed resource managers. Moreover, as outlined in Section 2.2.3, batch scheduling systems systems fail to keep up with the latest developments in distributed computing, e. g., (i) by omitting support for data parallelism, (ii) by storing data in a central location, or (iii) by denying workflow schedulers the control over resources required to implement data locality and exploit heterogeneity of distributed resources.

Furthermore, despite considerable efforts of the scientific workflow community to facilitate the sharing of workflows in public repositories (Goble and de Roure, 2007), true reproducibility of scientific experiments is thwarted since the provisioning of input data and setup of the execution environment is disregarded in most workflow management systems (Gil et al., 2007; Santana-Perez et al., 2014). Finally, most scientific workflow management systems support only static, acyclic workflow graphs, disallowing data-dependent programmatic concepts like conditionals and recursions, thereby unnecessarily limiting the flexibility of scientific workflows. While the scientific workflow community is becoming increasingly aware of these issues (e. g., Zhao et al., 2015; Santana-Perez et al., 2014; Brandt et al., 2015), to date only isolated, often domain-specific solutions addressing only subsets of these problems have been proposed (e. g., Amstutz et al., 2016; Di Tommaso et al., 2015, 2017).

Conversely, recently developed distributed dataflow systems, such as Spark (Zaharia et al., 2010) or Flink (Alexandrov et al., 2014) support distributed resource managers such as Hadoop YARN and provide highly scalable alternatives for implementing and executing data-intensive analysis pipelines. However, as outlined on page 22, such systems

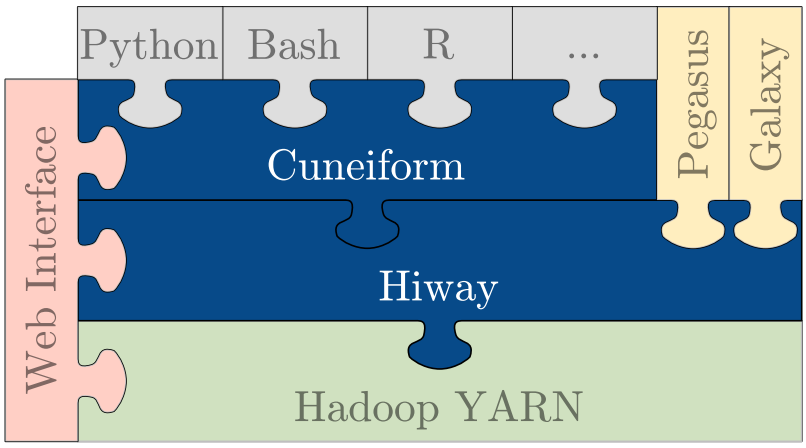


Figure 5.1: The SAASFEE software stack, comprising the Cuneiform workflow language and the Hi-WAY execution engine.

employ a semi-structured white-box (e. g., key-value-based) data model to be able to automatically partition and parallelize dataflows. Unfortunately, a structured data model impedes the flexibility in workflow design when integrating external tools that read and write file-based data. To circumvent this problem, additional glue code for transforming to and from the structured data model has to be provided. This introduces unnecessary overhead in terms of time required for implementing the glue code as well as for data transformations at runtime (Wu et al., 2016).

In this chapter, we describe the scientific workflow execution engine Hi-WAY[1]. Hi-WAY is part of the scientific workflow management system SAASFEE[2] (see Figure 5.1). Technically, Hi-WAY is an application master for Hadoop YARN that is able to interpret and execute scientific workflow specifications expressed in different languages. It emphasizes data center compatibility by being able to run on Hadoop installations of any size and type of underlying infrastructure. Compared to established scientific workflow management systems, Hi-WAY brings the following specific features, many of which were also recently identified as being critical for the future of scientific workflows (Deelman et al., 2017).

1. *Performance* gains through *adaptive* scheduling. Hi-WAY implements the adaptive workflow scheduler ERA presented in Section 4.2. It utilizes statistics of earlier workflow executions for estimating the runtimes of tasks awaiting execution. Using these runtime estimates, Hi-WAY is able to exploit heterogeneity in the computational infrastructure, adapt to variations of performance at runtime, and exhibit robustness in the face of straggler machines and failed task executions. Furthermore, Hi-WAY supports an array of alternative scheduling policies for different use cases (see Section 5.1.4).

[1]The code of Hi-WAY is available at `https://github.com/marcbux/Hi-WAY`
[2]`http://saasfee.io/`

2. *Scalable* execution. By employing Hadoop YARN and HDFS for distributed resource management and data storage, Hi-WAY harnesses its proven scalability and fault tolerance (see Section 5.1.1).

3. *Multi-language* support. Hi-WAY employs a generic yet powerful execution model. It has no own specification language, but instead comes with an extensible language interface and built-in support for multiple workflow languages. Supported languages include Cuneiform (Brandt et al., 2015, 2017), Pegasus DAX (Deelman et al., 2015), and Galaxy (Goecks et al., 2010) (see Section 5.1.2).

4. *Iterative* workflows. Hi-WAY's execution model supports data-dependent control-flow decisions. This allows for the design of conditional, iterative, and recursive structures, which are increasingly common in distributed dataflows (e. g., Murray et al., 2011), and are also beginning to emerge in scientific workflows (Prodan and Fahringer, 2005) (see Section 5.1.3).

5. *Reproducible* experiments. Hi-WAY generates comprehensive provenance traces, which can be directly re-executed as workflows (see Section 5.1.5). Also, Hi-WAY uses Chef for specifying automated setups of a workflow's software requirements and input data, including (if necessary) the installation of Hi-WAY and Hadoop (see Section 5.1.6).

The remainder of this chapter is structured as follows: Section 5.1 presents the architecture of Hi-WAY and gives detailed descriptions of the aforementioned core features. Section 5.2 describes several experiments showcasing these feature in real-life workflows on both local clusters and cloud computing infrastructure. Section 5.3 gives an overview of related work and Section 5.4 provides a summary of this chapter.

5.1 Hiway

Hi-WAY utilizes Hadoop as its underlying system for the management of both distributed computational resources and storage. It comprises three main components, as shown in Figure 5.2. First, the Workflow Driver reads a scientific workflow specified in any of the supported (textual) workflow languages and reports discovered tasks to the Workflow Scheduler (see Sections 5.1.2 and 5.1.3). Secondly, the Workflow Scheduler assigns ready tasks to compute resources provided by Hadoop YARN according to a selected scheduling policy (see Section 5.1.4). Finally, the Provenance Manager gathers comprehensive statistics obtained during task and workflow execution, handling their long-term storage and providing the Workflow Scheduler with up-to-date statistics on previous task executions (see Section 5.1.5). Automated installation routines for the setup of Hadoop, Hi-WAY, and selected workflows are described in Section 5.1.6.

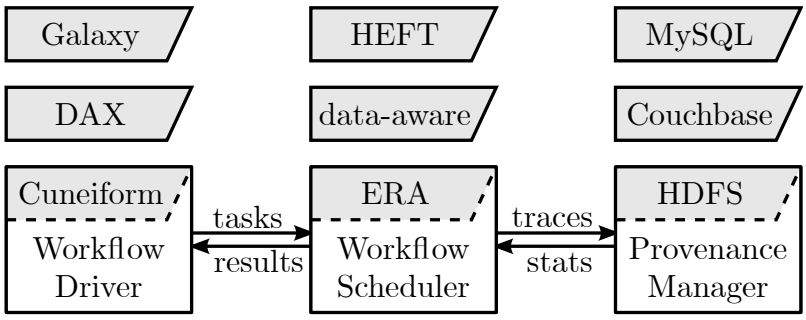


Figure 5.2: The architecture of the Hi-WAY application master: The Workflow Driver, described in Sections 5.1.2 and 5.1.3, reads a textual workflow file, monitors workflow execution, and notifies the Workflow Scheduler whenever it discovers new tasks. Tasks that are ready to be executed are assigned to computational resources by the Workflow Scheduler, which is presented in Section 5.1.4. Upon termination of a task, the Workflow Scheduler reports results to the Workflow Driver, which might determine new tasks as a consequence. Provenance and statistics data obtained during workflow execution are handled by the Provenance Manager (see Section 5.1.5) and can be stored in a local file as well as in a MySQL or Couchbase database.

5.1.1 Interface with Hadoop YARN

Hadoop version 2.0 introduced the resource management component YARN along with the concept of job-specific application masters (AMs), increasing scalability beyond thousands of computational nodes and enabling native support for non-MapReduce AMs (Vavilapalli et al., 2013). Hi-WAY seizes this concept by providing its own AM that interfaces with YARN (see Figure 5.3). See Section 2.2.3.2 for a description of Hadoop and its architectural concepts, namely YARN, its resource manager (RM) and node managers (NMs), as well as HDFS with its name node (NN) and data nodes (DNs).

To submit workflows for execution, Hi-WAY provides a light-weight client program. Each workflow that is launched from a client results in a separate instance of a Hi-WAY AM being spawned in its own container. In the case of multiple workflows running at the same time, the workload associated with workflow execution management is thereby distributed across multiple containers (and, possibly, multiple machines), which yields superior scalability over a centralized approach. The amount of memory and virtual cores of this container is specified in Hi-WAY's configuration.

For any of a workflow's tasks that await execution, the Hi-WAY AM responsible for running this particular workflow requests a worker container from YARN. Once allocated, the lifecycle of these worker containers involves (i) obtaining the task's input data from HDFS, (ii) invoking the command(s) associated with the task, and (iii) storing any generated output data in HDFS for consumption by other containers executing tasks in the future and possibly running on other compute nodes. Figure 5.4 illustrates this interaction between Hi-WAY's client application, AM and worker containers, as well as Hadoop's HDFS and YARN components.

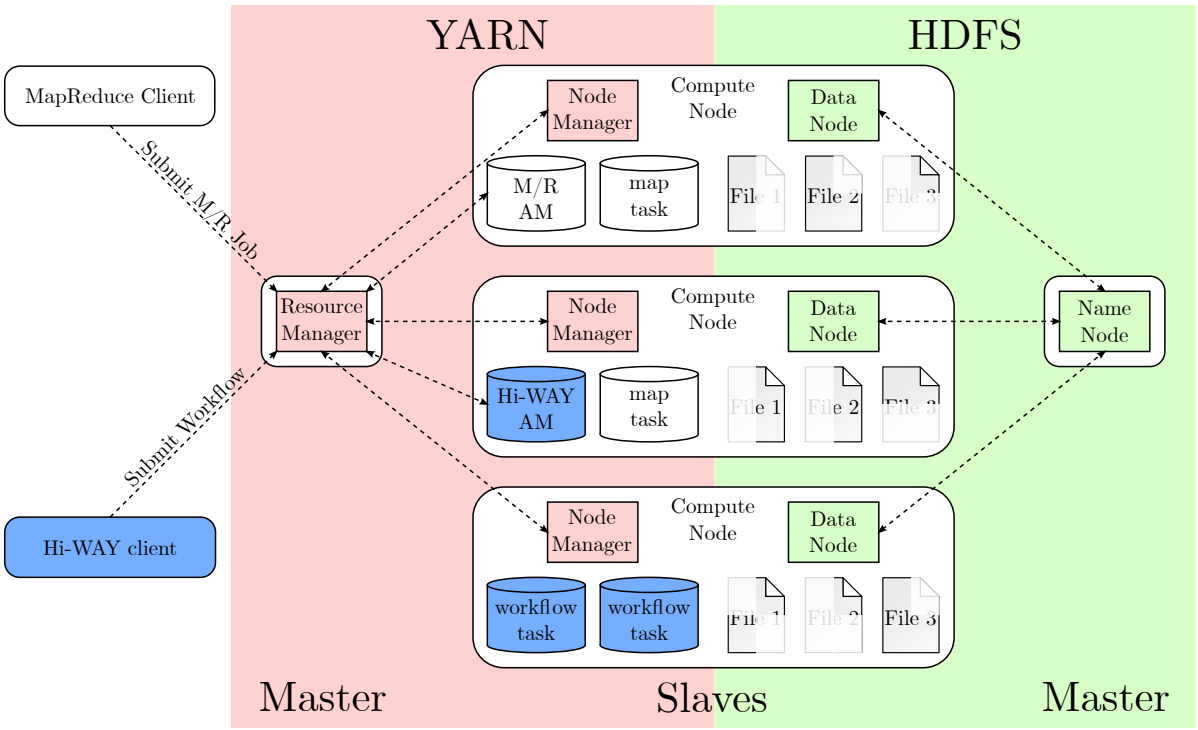


Figure 5.3: Architecture of a Hadoop version 2.x installation (comprising YARN and HDFS) to which both a Hi-WAY and a MapReduce job have been submitted. Processes of Hi-WAY are displayed in blue, whereas processes of YARN and HDFS are displayed in red and green, respectively. Clients of different programming models (e. g., scientific workflows or MapReduce) can submit jobs to YARN's RM. Upon receiving a new job, the RM negotiates with YARN's distributed NMs for a container to place an AM in. Each job is provided with its own separate AM that schedules and monitors execution. A scientific workflow job submitted to YARN results in a Hi-WAY AM operating from within its own container. From there, it negotiates with YARN's RM for additional resources (in the form of containers) required for workflow execution. Upon allocation of a new worker container, Hi-WAY selects a workflow task for execution within that container according to a selected scheduling policy (e. g., ERA). Any of the workflow's file-based data (input, intermediate, and output) is placed in HDFS. Data in HDFS is partitioned into blocks of equal size (128 MB by default) and blocks are replicated across distributed DNs (three replicas per block by default). HDFS's NN maintains the file index and can be interfaced with if data is to be read from or written to HDFS.

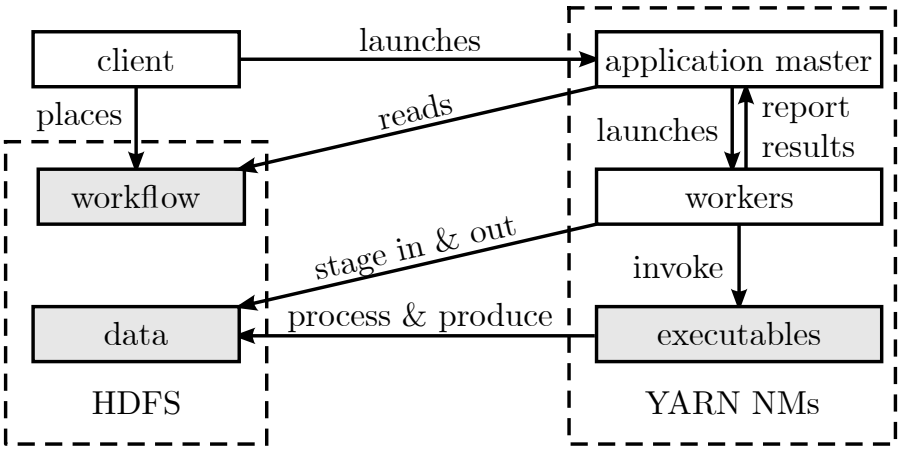


Figure 5.4: Interaction between the processes of Hi-WAY and Hadoop (white boxes; see Section 5.1.1) as well as files required for running a workflow (gray boxes; see Section 5.1.6). Workflow execution is initialized from a client application, resulting in a textual workflow file placed in HDFS and a new instance of a Hi-WAY AM launched within a container provided by one of YARN's node managers (NMs). This AM reads the workflow file residing in HDFS and prompts YARN to spawn worker containers for tasks that are ready to run. During task execution, these worker containers obtain input data from HDFS, invoke locally available executables reading this data, and generate output data, which is placed in HDFS for use by other worker containers. Results are also reported to the AM, which might lead to the discovery of new tasks.

A prerequisite for scalable workflow execution on commodity hardware with limited lifetime guarantees is the ability to recover from failures. To this end, Hi-WAY retries failed tasks, requesting YARN to allocate additional containers on different compute nodes. Also, data processed and produced by Hi-WAY persists through the crash of a storage node, since Hi-WAY uses the redundant file storage HDFS for any input, output, and intermediate data associated with a workflow.

5.1.2 Workflow Language Interface

Hi-WAY separates the tight coupling of scientific workflow languages and execution engines prevalent in established scientific workflow management systems. For this purpose, its Workflow Driver (see Section 5.1.3) provides an extensible, multilingual language interface, which is able to interpret scientific workflows written in a number of established workflow languages. Currently, four scientific workflow languages are supported: (i) the textual workflow language Cuneiform, (ii) DAX, the XML-based workflow language of the scientific workflow management system Pegasus, (iii) workflows exported from the scientific workflow management system Galaxy, and (iv) Hi-WAY provenance traces, which can also be interpreted as scientific workflows (see Section 5.1.5).

Cuneiform (Brandt et al., 2015, 2017) is a minimal workflow language that allows direct integration of code written in a range of external programming languages (e. g., Bash, Python, R, Perl, Java). It treats tasks as black boxes, allowing the integration of tools and libraries independent of their internal implementation. In contrast to most other workflow languages, the complete workflow graph cannot be determined or inferred prior to execution in Cuneiform. Instead, Cuneiform only allows for the determination of tasks at runtime when they are ready for immediate execution. While this prevents the use of static scheduling policies (see Section 2.2.4.2), it enables Cuneiform to support data-dependent iterative workflows, which may contain unbounded iterations, conditionals, and recursions. Cuneiform facilitates the assembly of highly parallel data processing pipelines by providing a range of second-order functions, including map and reduce operations.

DAX is Pegasus' built-in workflow description language, in which workflows are specified in an XML file (see Section 2.3.1.1). Contrary to Cuneiform, DAX workflows are static, explicitly specifying every task to be invoked and every file to be processed or produced by these tasks during workflow execution. Consequently, DAX workflows can become quite large and are not intended to be read or written by workflow developers directly. APIs enabling the generation of very large DAX workflows are provided for Java, Python, and Perl. For instance, in the case of bag-of-tasks workflows (see Definition 3 on page 9) these APIs facilitate the specification of tasks within the same bag by means of iteration (as opposed to having to separately and explicitly specify each task).

Workflows in the web-based scientific workflow management system Galaxy are created using a graphical user interface (see Section 2.3.2.3). Tasks comprising the workflow can be selected from a large range of software libraries that are part of any Galaxy installation. This process of workflow assembly results in a static workflow graph that can be exported to a JSON file, which can then be interpreted by Hi-WAY. In workflows exported from Galaxy, the workflow's input files are not explicitly designated. Instead, input ports serve as placeholders for the input files, which are resolved interactively when the workflow is committed to Hi-WAY for execution.

In addition to these workflow languages, Hi-WAY can easily be extended to interpret and execute other non-interactive workflow languages. For non-iterative languages, one only needs to extend the Workflow Driver class and implement the method that reads a textual workflow file to determine the tasks and data dependencies composing the workflow. In addition, for iterative languages one also has to implement the discovery of potential new tasks upon termination of a previous task.

5.1.3 Iterative Workflow Driver

On execution onset, the Workflow Driver reads the workflow file to determine executable tasks along with the files they process and produce (see last section for an overview of supported workflow languages). Any discovered tasks are passed to the Workflow Scheduler, which then assembles a schedule and creates container requests whenever a task is ready (see page 9 for definitions of blocked, ready, running, and completed tasks). Subsequently, the Workflow Driver supervises workflow execution, waiting for container

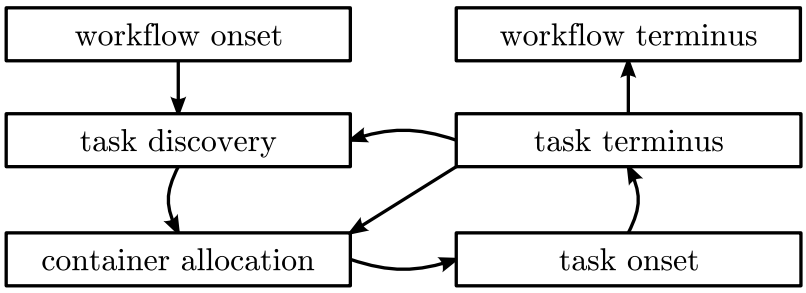


Figure 5.5: The iterative Workflow Driver's execution model. A workflow is read, entailing the discovery of tasks as well as the request for and eventual allocation of containers for ready tasks. Upon completion of a task executed in an allocated container, previously discovered tasks might become ready (resulting in new container requests), new tasks might be discovered, or the workflow might terminate.

requests to be fulfilled or for tasks to terminate. In the former case, the Workflow Driver requests the Workflow Scheduler to choose a task to be launched in that container. In the latter case, it registers any newly produced data, checks if new tasks have become ready, and potentially issues new container requests.

One of Hi-WAY's core strengths is its ability to interpret iterative workflows, which may contain data-dependent loops, conditionals, and recursive tasks (Prodan and Fahringer, 2005). In such workflows, the termination of a task may entail the discovery of entirely new tasks. For this reason, the Workflow Driver dynamically evaluates the results of completed tasks[3], forwarding newly discovered tasks to the Workflow Scheduler. See Figure 5.5 for an abstract visualization of the Workflow Driver's execution model.

As an example for an iterative workflow, consider an implementation of the k-means clustering algorithm commonly encountered in machine learning applications. k-means provides a heuristic for partitioning a (potentially very large) number of data points (e. g., intermediate data products of a bag-of-tasks workflow) into k clusters. To this end, over a sequence of parallelizable steps, an initial random clustering is iteratively refined until convergence is reached. The implementation of this algorithm requires the programmatical concepts of conditional task execution and unbounded iteration, which underlines the importance of such iterative control structures in scientific workflows.

5.1.4 Workflow Scheduler

The Workflow Scheduler is responsible for determining a suitable assignment of tasks to compute nodes. To this end, it receives tasks discovered by the Workflow Driver for which it creates container requests. Whenever a container has been allocated on a compute node, it selects a task for execution. This higher-level view on scheduling is different to YARN's internal schedulers, which, at a lower level, determine how to dis-

[3]In the case of Cuneiform workflows, the only iterative workflows currently supported by Hi-WAY, this dynamic evaluation is conducted by an imported Cuneiform library.

tribute resources between multiple users and applications. Hi-WAY provides a selection of workflow scheduling policies that optimize performance for different workflow structures and computational architectures.

Similar to all of the workflow management systems outlined in Section 2.3, Hi-WAY supports knowledge-free round-robin and FCFS scheduling. Furthermore, Hi-WAY can be configured to employ static HEFT scheduling like Pegasus. See Sections 2.2.4.1 and 2.2.4.2 for descriptions of these established and oft-encountered scheduling mechanisms.

When configured to employ a static scheduling policy, Hi-WAY's Workflow Scheduler assembles this schedule at the beginning of workflow execution and enforces containers to be placed on specific compute nodes according to this schedule. Since static schedulers like round-robin and HEFT require the complete graph structure of a workflow to be deductible at the onset of computation, static scheduling can not be used in conjunction with workflow languages that enable iterative workflows. Hence, HEFT scheduling is not compatible with Cuneiform workflows (see Section 5.1.3).

In addition to these schedulers, Hi-WAY also provides a data-aware scheduler intended for I/O-intensive workflows. The data-aware scheduler minimizes data transfer by assigning tasks to compute nodes based on the amount of input data that is already present locally. To this end, whenever a new container is allocated, the data-aware scheduler skims through all ready tasks. From these ready tasks, it then selects the task with the highest fraction of input data available locally (in HDFS) on the compute node hosting the newly allocated container.

By default, the amount of memory allotted to each task is uniform across all worker containers and can be configured in form of a parameter. While this allows Hi-WAY to make just-in-time scheduling decisions, it might lead to considerable overhead for workflows comprising only a few memory-intensive tasks alongside a large number of tasks with a small memory footprint. For this reason, Hi-WAY provides a memory-aware scheduling policy. By means of a configuration file in JSON format, this policy allows the specification of separate amounts of memory allotted to each bag of tasks (see Definition 3 on page 9).

In addition to these scheduling policies, Hi-WAY is also able to employ adaptive scheduling in which the assignment of tasks to compute nodes is based on continually updated runtime estimates and is therefore adapted to the computational infrastructure. To this end, Hi-WAY implements the ERA scheduler presented in Section 4.2 as its default scheduling policy. Whenever a new container is allocated, ERA determines a suitable bag of tasks to select a task from by means of its exploitation heuristic, which is described in Section 4.2.3. In its implementation in Hi-WAY, ERA can be configured to not just select any task from the determined bag of tasks, but, similar to the data-aware scheduling policy, select the task with the most available input data.

To supply the ERA and HEFT schedulers with runtime estimates, the Workflow Scheduler assembles separate Wiener process models for each combination of bag of tasks and compute node as described in Section 4.1. These Wiener process models are assembled on the basis of runtime measurements for (bags of) tasks across all workflow executions. Runtime measurements are provided by the Provenance Manager, which is responsible for gathering, storing, and providing provenance and statistics data (see next section).

5.1.5 Provenance Manager

The Provenance Manager surveys workflow execution and registers events at different levels of granularity:

1. It traces events at the workflow level, including the name of the workflow and its total execution time.

2. It logs events for each task, e. g., the commands invoked to spawn the task, its makespan, its standard output and error channels and the compute node on which it ran.

3. It stores events for each file consumed and produced by a task. This includes its size and the time it took to move the file between HDFS and the local file system.

All of this provenance data is supplemented with timestamps as well as unique identifiers and stored as JSON objects in a trace file in HDFS, from where it can be accessed by other instances of Hi-WAY.

Since this trace file holds information about all of a workflow's tasks and data dependencies, it can be interpreted as a workflow itself. Hi-WAY promotes reproducibility of experiments by being able to interpret and execute such workflow traces directly through its Workflow Driver, albeit not necessarily on the same compute nodes. Hence, workflow trace files generated by Hi-WAY constitute a fourth supported workflow language.

Evidently, the amount of workflow traces can become difficult to handle for heavily-used installations of Hi-WAY with thousands of trace files or more. To cope with such high volumes of data, Hi-WAY provides prototypical implementations for storing and accessing this provenance data in a MySQL or Couchbase database as an alternative to storing trace files in HDFS[4]. The usage of a database for storing this provenance data brings the added benefit of facilitating manual analysis and exploration (Schuh, 2015).

5.1.6 Reproducible Installation

The properties of the scientific workflow programming model with its black-box data and operator models as well as the usage of Hadoop for resource management and data distribution both dictate requirements for workflow designers (for an illustration of these requirements, see Figure 5.4). First, all of a workflow's software dependencies (executables, software libraries, etc.) have to be available on each of the compute nodes managed by YARN, since any of the tasks composing a workflow could be assigned to any compute node. Secondly, any input data required to run the workflow has to be placed in HDFS or made locally available on all nodes.

To set up an installation of Hi-WAY and Hadoop, configuration routines are available online in the form of Chef installation routines – so-called recipes. Chef is a configuration management software for the automated setup of computational infrastructures.

[4]Joint work with Hannes Schuh, who implemented the database connectors.

Hi-WAY's Chef recipes allow for the setup of standalone or distributed Hi-WAY installations, either on local machines or in public compute clouds such as Amazon EC2. In addition, recipes are available for setting up a large variety of execution-ready workflows, including the workflows described in Sections 2.1.1 and 2.1.2. This includes obtaining the workflows' input data, placing them in HDFS, and installing any software dependencies required to run the workflows.

Besides providing a broad array of use cases, these recipes enable reproducibility of the experiments outlined in Section 5.2. The procedure of running these Chef recipes via the orchestration engine Karamel[5] to set up a distributed Hi-WAY execution environment along with a selection of workflows is described on `http://saasfee.io` in the form of textual descriptions and video tutorials. These Chef recipes allow for reproducible experiments without having to provide virtual machine images, but merely by means of configuration routines, as proposed by Santana-Perez et al. (2014).

Note that this means of providing reproducibility exists in addition to the executable provenance traces described in Section 5.1.5. While the Chef recipes are well-suited for reproducing experiments across different research groups and compute clusters, the executable trace files are intended for use on the same cluster. The reason for this is that running a trace file requires input data to be located and software requirements to be available just like during the workflow run from which the trace file was derived.

5.2 Evaluation

We conducted a number of experiments in which we evaluated Hi-WAY's core features of scalability, performance, and adaptive workflow scheduling. The remaining properties (support for multilingualism, reproducible experiments, and iterative workflows) are achieved by design. The workflows used in this section are written in three different languages and – apart from the experiment in Section 5.2.4 – can be automatically set up (including input data) and run on Hi-WAY with only a few clicks following the procedure described in Section 5.1.6.

Across the experiments described here, we executed relevant workflows from different areas of research on both virtual clusters of Amazon EC2 and on local computational infrastructure. Section 5.2.1 outlines two experiments in which we analyze the scalability and performance behavior of Hi-WAY when increasing the number of available computational nodes to very large numbers. In Section 5.2.2, we describe an experiment that contrasts the performance of running a computationally intensive Galaxy workflow on both Hi-WAY and Galaxy. In Section 5.2.3 we report on an experiment in which the potential gains in performance achievable through adaptive scheduling are evaluated. Finally, in Section 5.2.4, we examine diminishing returns of hardware investments by running a large-scale workflow employed in practice on computational infrastructures with acquisition costs of different orders of magnitude.

Table 5.1 gives an overview of all experiments described in this section.

[5] `http://www.karamel.io/`

Table 5.1: Overview of conducted experiments, their evaluation goals and the section in which they are outlined.

workflow	language	scheduler	infrastructure	runs	evaluation	section
SNV calling	Cuneiform	data-aware	24 Xeon E5-2620	3	scalability, perf.	5.2.1
SNV calling	Cuneiform	FCFS	128 EC2 m3.large	3	scalability	5.2.1
RNA-seq	Galaxy	data-aware	6 EC2 c3.2xlarge	5	performance	5.2.2
Montage	DAX	HEFT, ERA	8 EC2 m3.large	80	adaptive scheduling	5.2.3
SNV calling	Cuneiform	FCFS	24 Xeon E5-2620	1	scalability	5.2.4

5.2.1 Scalability

For evaluating the scalability of Hi-WAY, we employed a SNV calling workflow as described in Section 2.1.1.1. The input of this workflow were genomic reads obtained from the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015). Reads were mapped against a reference genome using the tool Bowtie 2 (Langmead and Salzberg, 2012), mappings were sorted using SAMtools (Li et al., 2009), and genomic variants were determined using VarScan (Koboldt et al., 2009). Finally, detected variants were annotated using the ANNOVAR (Wang et al., 2010) toolkit.

In a first experiment we implemented this workflow in both Cuneiform and Apache Tez (see Section 2.2.3.2 for a description of Tez)[6]. We ran both Hi-WAY and Tez on a Hadoop installation on a local cluster comprising 24 compute nodes connected via a one gigabit switch. Each compute node provided 24 gigabyte of memory as well as two Intel Xeon E5-2620 processors with 24 virtual cores combined. This resulted in a maximum of 576 concurrently running containers, of which each one was provided with its own virtual processor core and one gigabyte of memory. The data-aware scheduler was selected as Hi-WAY's scheduling policy[7].

The results of this experiment are illustrated in Figure 5.6. Scalability beyond 96 containers was limited by network bandwidth. The results indicate that Hi-WAY performs comparably to Tez as long as network resources are sufficient, yet, due to its data-aware scheduling policy, scales favorably when network resources are limited. The data-aware scheduling policy reduced data transfer by preferring to assign the data-intensive reference mapping tasks to containers on compute nodes with a locally available replica of the input data. Finally, another important finding of this experiment was that the implementation of the workflow in Cuneiform resulted in very little code and was finished in a few days, whereas it took several weeks and a lot of code in Tez (Lipka, 2014).

The reason for network bandwidth being the scalability bottleneck in this experiment was that both the genomic reads and the reference, which served as the workflow's input data, were split into small data fragments of a few megabytes Consequently, reference mapping tasks were comparably short-running, which, as described in Section 2.2.1.2,

[6]Joint work with Carsten Lipka, who implemented and executed the workflow in Apache Tez.

[7]The data-aware scheduler was Hi-WAY's default scheduling policy at the time as ERA was not implemented yet.

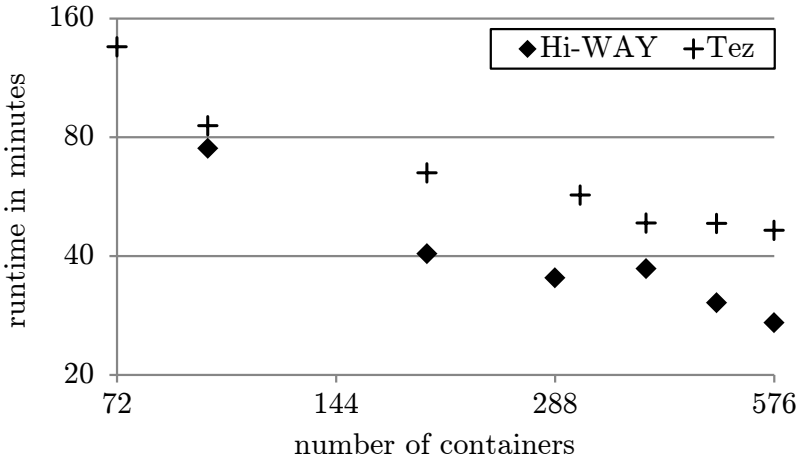


Figure 5.6: Mean runtimes of the SNV calling workflow with increasing number of containers. Note that both axes are in logarithmic scale.

can lead to substantial I/O overhead for tasks with multiple input data. In a second experiment, in which we evaluated Hi-WAY's scalability on real cloud infrastructure, we circumvented these issues by increasing the volume and fragment size of input data while at the same time reducing network load. Specifically, we (i) used additional genomic read files from the 1000 Genomes Project, (ii) compressed mappings using CRAM referential compression (Li et al., 2009), and (iii) obtained input read data during workflow execution from the Amazon S3 bucket of the 1000 Genomes Project, which only added a very minor overhead over having read data available on the cluster in HDFS. To evaluate Hi-WAY's scalability in isolation of the selected scheduling policy, we configured Hi-WAY to employ FCFS scheduling. Other than that, both Hi-WAY and Hadoop were set up with its default parameters.

In the process of this second experiment, the workflow was first run using a single worker node, processing a single genomic sample comprising eight files, each about one gigabyte in size. In subsequent runs, we repeatedly doubled the number of worker nodes and the volume of input data. In the last run (after seven duplications), the computational infrastructure consisted of 128 worker nodes, whereas the workflow's input data comprised 128 samples of eight roughly gigabyte-sized files each. Hence, this procedure resulted in a total volume of more than a terabyte of data.

The experiment was run three times on virtual clusters of Amazon EC2. To investigate potential effects of datacenter locality on workflow runtime (which we did not observe during the experiment), these clusters were set up in different EC2 regions – once in the EU West (Ireland) and twice in the US East (North Virginia) region. Since we intended to analyze the scalability of Hi-WAY, we isolated the Hi-WAY AM from the worker threads and Hadoop's master threads. To this end, dedicated compute nodes were provided for (i) the Hi-WAY AM, running in its own YARN container, and (ii) the two Hadoop master threads (RM and NN). All compute nodes – the two master nodes and all of the up to 128 worker nodes – were configured to be of type m3.large, each providing two virtual processing cores, 7.5 gigabytes of main memory, and 32 gigabytes of local SSD storage.

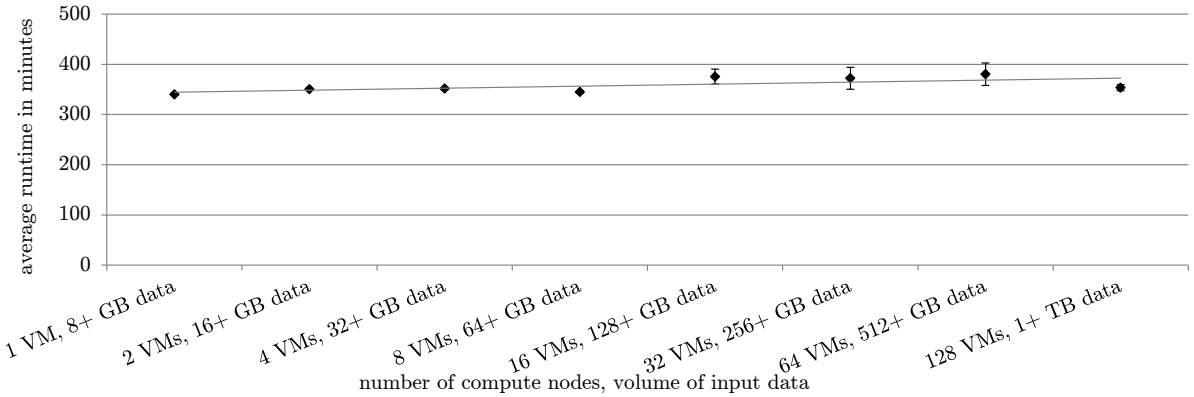


Figure 5.7: Mean runtimes for three runs of the SNV calling workflow described in Section 5.2.1 when repeatedly doubling the number of compute nodes available to Hi-WAY along with the input data to be processed. The error bars represent the standard deviation, whereas the line represents the (linear) regression curve. The standard deviation is higher for cluster sizes of 16, 32, and 64 nodes, which is due to the observed runtime of the CPU-bound variant calling step being notably higher in one run of the experiment. Since these three measurements were temporally co-located and we did not observe similar distortions at any other point in time, this observation can most likely be attributed to external factors.

All of the experiment runs were set up automatically using Karamel (see Section 5.1.6). Over the course of the experiment we determined the runtime of the workflow. Furthermore, the CPU, I/O, and network performance of the master and worker nodes were monitored using the Linux tools *uptime*, *ifstat*, and *iostat*. Since the workflow's tasks required the whole memory available on a single compute node, we configured Hi-WAY to only allow a single container per worker node at the same time, enabling multithreading for tasks running within that container whenever possible.

The average of measured runtimes with steadily increasing amounts of both compute nodes and input data is displayed in Figure 5.7 and Table H in the Appendix. The regression curve indicates near-linear scalability: Doubling of input data and the associated doubling of workload is almost fully offset by a doubling of worker nodes. This is even true for the maximum investigated cluster size of 128 nodes, in which a terabyte of genomic reads were mapped and analyzed against the whole human genome. Note that extrapolating the average runtime for processing eight gigabytes of data on a single machine reveals that mapping a whole terabyte of genomic read data against the whole human genome along with further downstream processing would easily take a month on a single machine.

We identified several potential bottlenecks when scaling out a Hi-WAY installation beyond 128 nodes. For instance, Hadoop's master processes (RM and NN) could prove to limit scalability. Similarly, the Hi-WAY AM process that handles the scheduling of tasks, the assembly of results, and the tracing of provenance, could collapse when further

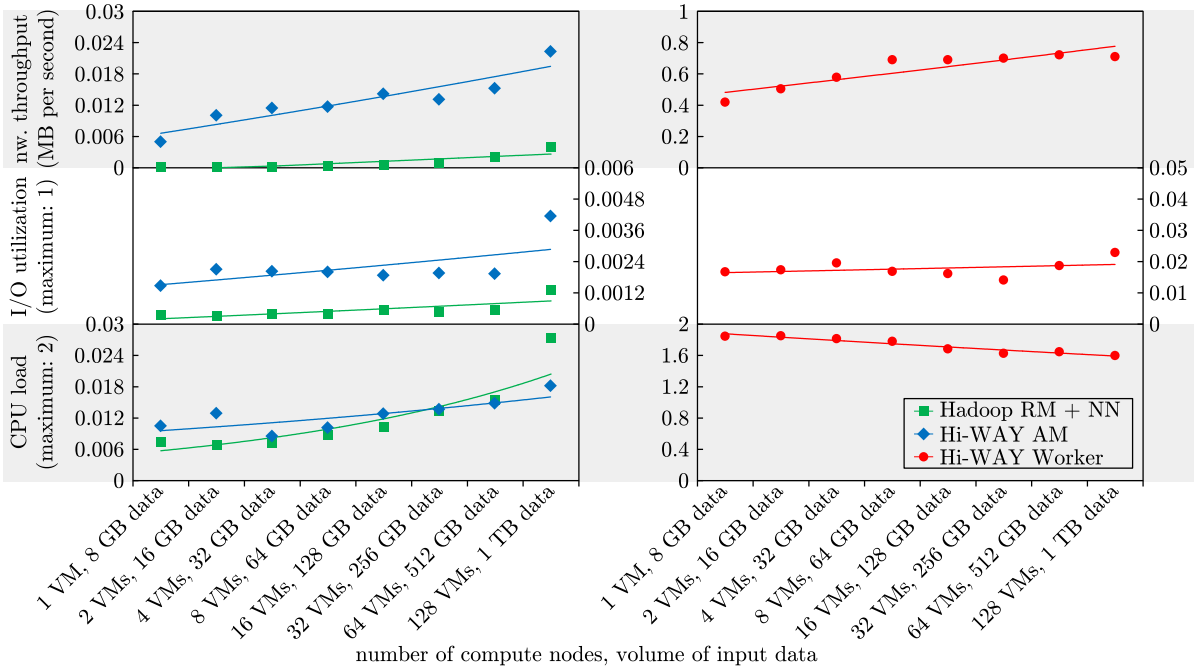


Figure 5.8: Resource utilization (CPU load, I/O utilization, and network throughput) of virtual machines hosting the Hadoop master processes, the Hi-WAY AM and a Hi-WAY worker process. Average values over the time of workflow execution and across experiment runs are shown along with their exponential regression curve. We observed the following peak values for worker nodes: 2.0 for CPU load (due to two virtual processing cores being available per machine), 1.0 for I/O utilization (since 1.0 corresponds to device saturation, i. e., 100 % of CPU time spent for I/O requests), and 109.35 MB per second for network throughput. Note the different scales for the master nodes on the left and the worker nodes on the right.

increasing the workload and the number of available compute nodes. To this end, we were interested in the resource utilization of these potential bottlenecks, which is displayed in Figure 5.8.

We observe a steady increase in load across all resources for the Hadoop and Hi-WAY master nodes when repeatedly doubling the workload and number of worker nodes. However, resource load stays well below maximum utilization at all cluster sizes. In fact, all resources are still utilized less than 5 % even when processing one terabyte of data across 128 worker nodes. Furthermore, we observe that resource utilization for Hi-WAY's master process is of the same order of magnitude as for Hadoop's master processes, which have been developed to scale to 10,000 compute nodes and beyond (Vavilapalli et al., 2013).

While resource utilization on the master nodes increases when growing the workload and computational infrastructure, we observe that CPU utilization stays close to the maximum of 2.0 on the worker nodes, whereas the other resources stay under-utilized.

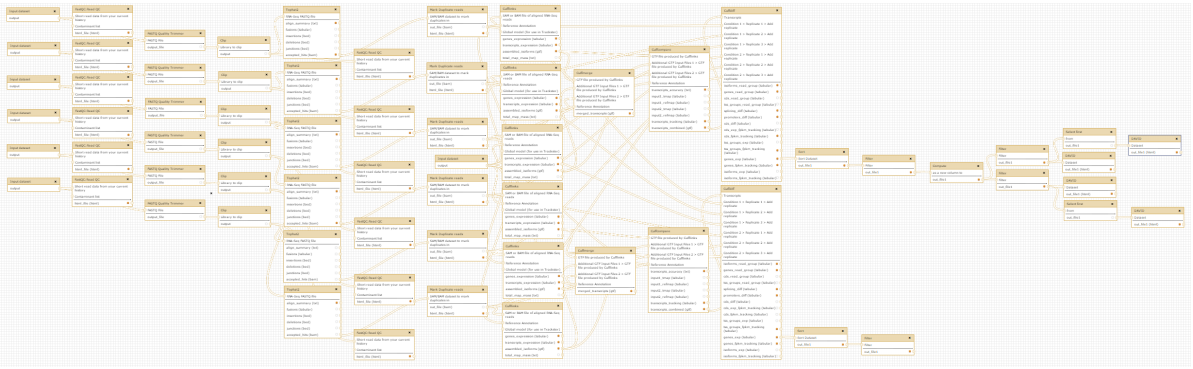


Figure 5.9: Screenshot of the TRAPLINE workflow on the public Galaxy server.

This finding is unsurprising, since both the mapping step and the variant calling step of the workflow support multithreading and are CPU-bound. Hence, this finding confirms that the workers are nearly fully utilized for processing the workflow, whereas the master processes appear to be able to cope with a considerable amount of additional load.

5.2.2 Performance

Wolfien et al. (2016) presented a Galaxy-based implementation of the RNA sequencing workflow described in Section 2.1.1.2. The workflow is available through Galaxy's public workflow repository. Their implementation of the workflow, called TRAPLINE, compares two genomic samples. Each of these two samples is expected to be available in triplicates and the majority of data processing tasks composing the workflow are arranged in sequential order. For these reasons, the workflow natively has a degree of parallelism of six across most of its parts. See Figure 5.9 for a screenshot of the TRAPLINE workflow.

We exported the TRAPLINE workflow from Galaxy and ran it on virtual clusters of Amazon EC2 using both Hi-WAY and Galaxy. Virtual clusters were set up automatically in Amazon's US East (North Virginia) region using Karamel for Hi-WAY (see Section 5.1.6) and CloudMan for Galaxy (see Section 2.3.2.3). All clusters consisted of compute nodes of type c3.2xlarge, providing eight virtual processing cores, 15 gigabytes of main memory and 160 gigabytes of local SSD storage each. Due to the workflow's degree of parallelism of six, we ran the workflow on clusters of sizes one up to six. For each cluster size, we executed this Galaxy workflow five times on Hi-WAY, comparing the average runtime against an execution on Galaxy CloudMan.

As workflow input data, we used RNA-seq data of young versus aged mice, obtained from the Gene Expression Omnibus (GEO) repository[8]. Input data obtained this way amounted to more than ten gigabytes in total. Since several tasks in TRAPLINE require large amounts of memory, we configured both Hi-WAY as well as CloudMan's default underlying batch scheduling system, Slurm, to only allow execution of a single task per worker node at any time. Omitting this configuration would cause either of the

[8]series GSE62762, samples GSM15330[14|15|16|45|46|47]

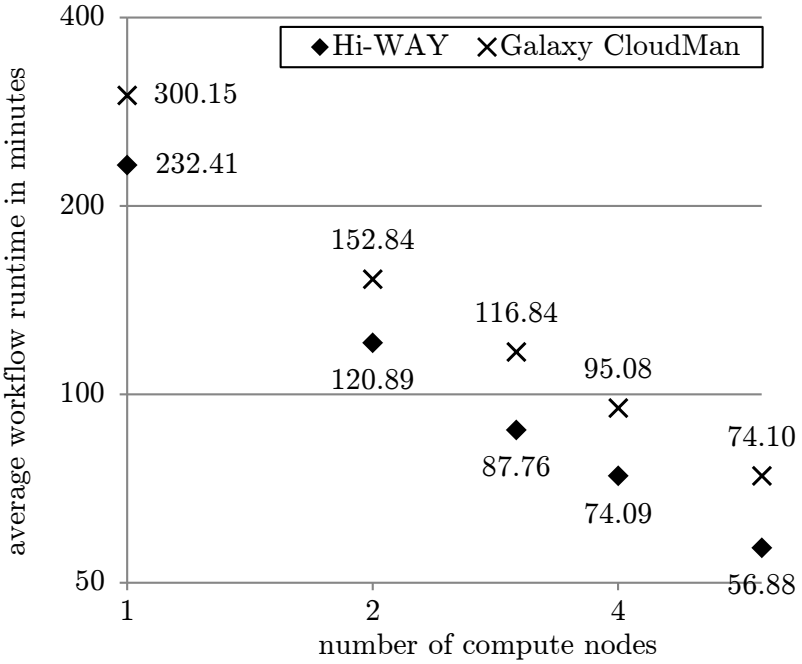


Figure 5.10: Average runtime of executing the TRAPLINE workflow described in Section 5.2.2 on Hi-WAY and Galaxy. The number of EC2 compute nodes of type c3.2xlarge was increased from one up to six. Note that both axes are in logarithmic scale.

two systems to run out of memory at some point during workflow execution. Other parameters were left unchanged from their default values[9].

The results of executing the TRAPLINE workflow on both Hi-WAY and Galaxy CloudMan are displayed in Figure 5.10. Across all of the tested cluster sizes, we observed that Hi-WAY outperformed Galaxy by at least 25 %. These differences were found to be significant by means of a one sample t-test (p-values of 0.000127 and lower).

The observed difference in performance is most notable in the computationally costly TopHat 2 step, which makes heavy use of multithreading and generates large amounts of intermediate files. Therefore, our speedup can be attributed to Hi-WAY utilizing the worker nodes' transient local SSD storage, since both Hadoop's distributed file system HDFS as well as the storage of YARN containers reside in the local file system. Conversely, Galaxy stores all of its data on an Amazon Elastic Block Store (EBS) volume, a persistent drive that is accessed over the network and shared among all compute nodes[10].

Apart from the observed gap in performance, it is important to point out that Galaxy CloudMan only supports the automated setup of virtual clusters of up to 20 nodes. Compared to Hi-WAY, it therefore only provides very limited scalability. We conclude

[9]Hi-WAY employed its data-aware scheduling policy, which, due to ERA not being implemented yet, was its default scheduling policy at the time.

[10]While EBS continues to be CloudMan's default storage option, a recent update has introduced support for using transient storage instead.

that Hi-WAY leverages the strengths of Galaxy, which lie in its intuitive means of workflow design and vast number of supported tools. In particular, Hi-WAY provides a more performant, flexible, and scalable alternative to Galaxy CloudMan for executing data-intensive Galaxy workflows with a high degree of parallelism.

5.2.3 Adaptive Scheduling

To showcase the benefits of adaptive scheduling on heterogeneous computational infrastructures, an experiment was performed in which we generated a Montage workflow as described in Section 2.1.2. The implementation of the workflow used in this experiment assembles a 0.4 degree mosaic image of the Omega Nebula. This resulted in a comparably small workflow with a maximum degree of parallelism of 21 during the image projection and background radiation correction phases. In the experiment, this workflow was repeatedly executed on a Hi-WAY installation set up on a virtual cluster in the EU West (Ireland) region of Amazon EC2. The cluster comprised a single master node as well as eleven worker nodes, which, similar to the scalability experiment in Section 5.2.1, were of type m3.large, providing two virtual cores and 7.5 GB of main memory. To fully utilize the virtual cores provided by these machines and match the workflow's degree of parallelism, worker nodes were configured to provide two task slots, i. e., to allow for the concurrent execution of up to two tasks.

To simulate a heterogeneous and potentially shared computational infrastructure, synthetic load was introduced on these machines by means of the Linux tool *stress*. To this end, only one worker machine was left unperturbed. Five worker machines were taxed with increasingly many CPU-bound processes and five other machines were impaired by launching increasingly many (in both cases 1, 4, 16, 64, and 256) processes writing data to the local disk.

A single run of the experiment, of which 80 were conducted in total, encompassed (i) running the Montage workflow once using a FCFS scheduling policy, which served as a baseline to compare against, (ii) running the workflow 20 times consecutively using the HEFT scheduler, and (iii) running the workflow 20 times consecutively using the ERA scheduler with ρ set to 2 (to match the number of task slots per virtual machine).

Evidently, both HEFT and ERA base their scheduling decisions on runtime estimates for all bag-of-task-machine-pairs. To this end, ERA obtained runtime estimates by means of its adaptation heuristic with α set to 0.2, as described in Section 4.2.1. Conversely, HEFT was provided with runtime estimates based on the last measured runtime of the same bag-of-task-machine-pair. Similar to ERA, if no such measurement was available, HEFT was provided with a runtime estimate of zero.

In the process of the 20 consecutive HEFT and ERA runs, larger and larger amounts of provenance data became available as a consequence of prior workflow executions. Hence, Hi-WAY's Workflow Scheduler was provided with increasingly comprehensive runtime estimates. However, whenever the scheduling policy was changed (i. e., after 20 iterations), all gathered task runtime statistics and assembled Wiener process models were discarded.

The results of this experiment are illustrated in Figure 5.11. The performance of HEFT scheduling improves with more and more provenance data becoming available.

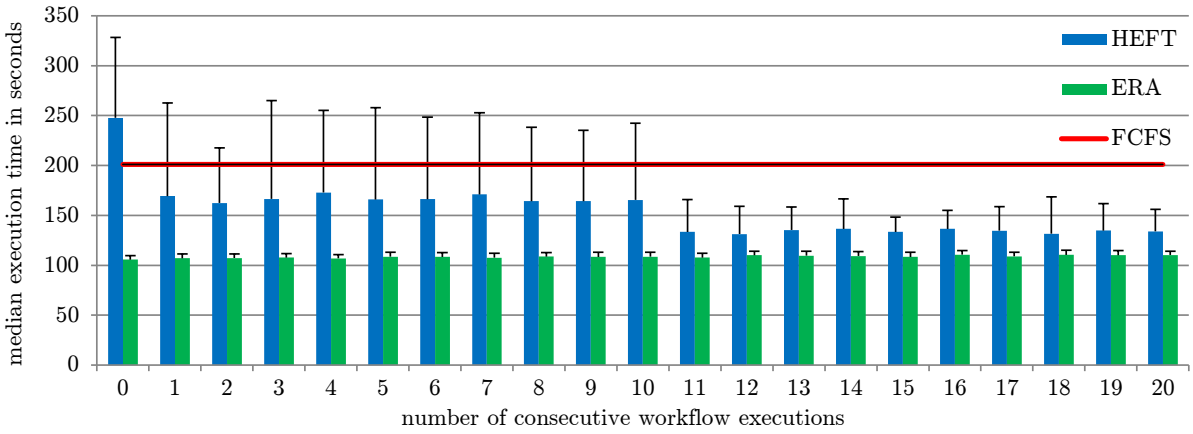


Figure 5.11: Median runtime of executing Montage on a heterogeneous infrastructure when using HEFT and ERA scheduling and increasing the number of previous workflow runs and thus the amount of available provenance data. The error bars represent the standard deviation.

Employing HEFT scheduling in the absence of any available provenance data results in subpar performance compared to FCFS scheduling. This is due to HEFT being a static scheduling policy, which entails that task assignments are fixed, even if one worker node still has many tasks to run while another, possibly more performant worker node is idle.

However, with a single prior workflow execution, HEFT already outperforms FCFS significantly (t-test, p-value of 0.033). The next significant performance gain can then be observed between ten and eleven prior workflow execution (p-value of $6.22 \cdot 10^{-7}$). At this point, any task composing the workflow, even the ones that are only executed once per workflow run, have been executed on all eleven worker nodes at least once. Hence, runtime estimates are complete and scheduling is no longer driven by the need to test additional task-machine-assignments. Note that this also leads to more stable workflow runtimes, which is reflected in a major reduction of the standard deviation of runtime.

While HEFT requires eleven prior workflow executions to reach its full potential, ERA already significantly outperforms FCFS without any prior runtime measurements (t-test, p-value of $1.34 \cdot 10^{-19}$). In addition, the makespans of workflows executed by ERA for the first time are significantly lower than the makespans of workflows scheduled by HEFT after 20 prior executions (p-value of $9.13 \cdot 10^{-29}$). Also, the variance in observed runtime is very small for workflows scheduled by ERA as opposed to the other two scheduling policies.

These finding can mostly be attributed to (i) ERA being a just-in-time scheduling policy that does not rely on static schedules, and, most importantly (ii) ERA's replication heuristic, which remedies the notable performance degradation when assigning tasks to the most heavily I/O-stressed compute node. ERA's exploitation and adaptation heuristics played a less important role in this experiment, since (i) the workflow's limited degree of parallelism and linear structure in between bags of tasks leaves little room for exploiting heterogeneity and (ii) the computational infrastructure, while

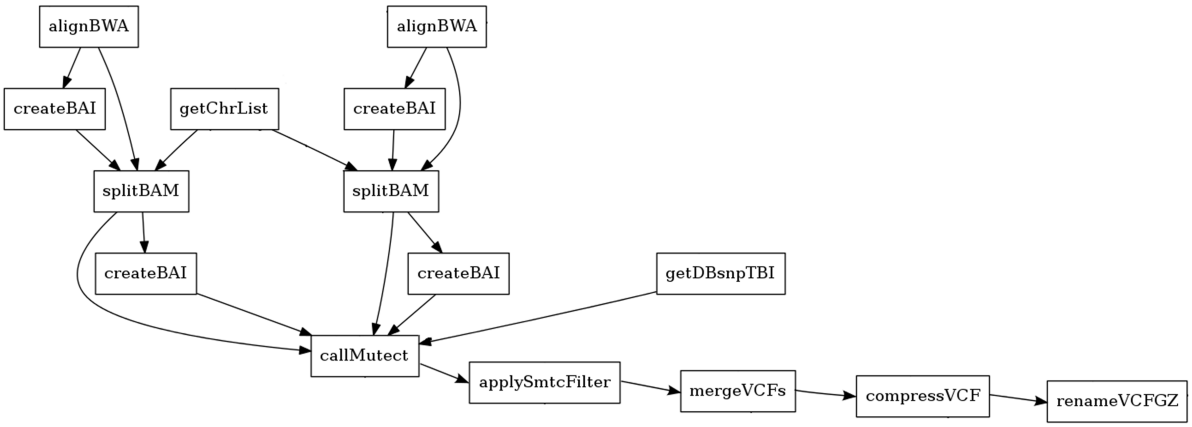


Figure 5.12: An abstractly visualized Cuneiform implementation of a SNV calling workflow utilized in practice at the Berlin Institute of Health. This graphical workflow representation was automatically generated by the Cuneiform workflow development toolkit.

very heterogeneous, was relatively stable. Note that we observed a slight increase in makespan variance if ERA's ρ parameter was set to 1, since in this setting, the single allowed replicate could be co-located on the same machine with its original task (results not shown).

5.2.4 Diminishing Returns on Hardware Investment

In this experiment, we examined the monetary cost of hardware investments and its returns in performance gained. To this end, we executed a data-intensive scientific workflow utilized in practice on computational infrastructures with largely varying acquisition costs. We then compared measured workflow runtimes against acquisition costs to get a rough indicator for the expected performance and data analysis throughput per invested money.

As evaluation workflow, we employed a SNV calling workflow, as outlined in Section 2.1.1.1. This implementation of the workflow uses the tools BWA, SAMtools, and MuTect for reference mapping and variant calling and queries dbSNP for variant characterization. The workflow was assembled using the workflow management system Snakemake (see Section 2.3.1.3) and re-implemented in the Cuneiform workflow language by mirroring tasks and data dependencies[11]. A visualization of this re-implemented Cuneiform workflow is shown in Figure 5.12.

Both Hi-WAY and Snakemake were configured with their default parameters. However, to minimize the effects of the selected scheduling policy on measured performance, we selected FCFS scheduling as Hi-WAY's scheduling policy. We ran the workflow on three different computational infrastructures which were acquired in a similar time frame:

[11] Joint work with the Core Unit Bioinformatics (CUBI) of the Berlin Institute of Health, who implemented and ran this workflow in Snakemake and Christopher Schiefer, who implemented and ran this workflow in Cuneiform.

1. *sonic*: The workflow was run using standalone Cuneiform (i. e., a non-distributed installation without Hi-WAY) on a monolithic server similar to the reference machine described in Section 3.4.1. Specifically, this machine provided four Intel Xeon E7-4870 processors, amounting to a total number of 80 virtual cores and 500 gigabytes of main memory. The acquisition costs of this machine were 11,535 € in the year 2014.

2. *dbis*: The workflow was executed using SAASFEE (i. e., Cuneiform, Hi-WAY, and Hadoop 2.7.1) on the distributed shared-nothing cluster described in Section 5.2.1. This cluster comprised 24 compute nodes, each providing 24 to 32 gigabyte of main memory and two Intel Xeon E5-2620 processors with 24 virtual cores. Its acquisition costs were 94,719 € in the year 2014.

3. *rubix*: The Snakemake implementation of the workflow was run using a Snakemake installation on a large-scale shared-nothing cluster providing 111 compute nodes with a total of 3784 cores. Each of the compute nodes in this cluster provided an amount of main memory equal to 128, 188, 500, or 1000 gigabytes. The cluster was acquired in 2014. Acquisition costs were estimated at around two million €.

The results of the experiment as shown in Table 5.2 and Figure 5.13 indicate that to achieve a doubling in throughput, one has to invest the quadruple acquisition cost. Reasons for this non-linear scaling of acquisition cost and data analysis throughput are manifold, including (i) overhead introduced through distribution and synchronization of resources and (ii) a smaller market niche and, thus, smaller production batch sizes for high-end hardware configurations (Geist and Reed, 2017).

Clearly, the reliability and amount of data these conclusions are based on should be improved, since (i) we only measured workflow execution time once per hardware configuration (to cut cost and occupancy of rubix, a production cluster), (ii) we merely had an estimate for the acquisition cost of the rubix infrastructure to work with, and (iii) we only presented results for a single application. Furthermore, we did not compare the potential overhead introduced by scientific workflow management systems SAASFEE and Snakemake (which, based for instance on the observed utilization of worker machines in Figure 5.8, we expect to be of minor impact).

5.3 Related Work

Projects with goals similar to Hi-WAY can be separated into two groups. The first group of systems comprises scientific workflow management systems, which, like Hi-WAY, employ black-box data and operator models. The second group encompasses distributed dataflow systems developed to process mostly structured or semi-structured (white-box) data. For overviews of both groups of systems, readers are referred to Sections 2.3 and 2.2.3, respectively. The remainder of this section concerns itself with (i) the differences between Hi-WAY and other scientific workflow management systems that support distributed computation and (ii) related systems and approaches not discussed in Sections 2.3 or 2.2.3.

Table 5.2: Workflow execution time of a SNV calling workflow employed in practice. The workflow was executed once on different hardware configurations of varying acquisition cost. Notably, the data analysis throughput does not increase linearly with acquisition cost. *Throughput measured in executions per year.

	sonic	dbis	rubix
type	monolithic	cluster	cluster
workflow system	Cuneiform	SAASFEE	Snakemake
compute nodes	1	24	111
virtual cores	80	576	3784
memory per node	512 GB	24 / 32 GB	124 / 188 / 500 / 1000 GB
acquisition cost	11,535 €	94,719 €	2,000,000 € (estimated)
execution time	24 hours	7.62 hours	1.14 hours
throughput*	365	1149.61	7684.21
throughput per €	0.0316	0.0121	0.0038

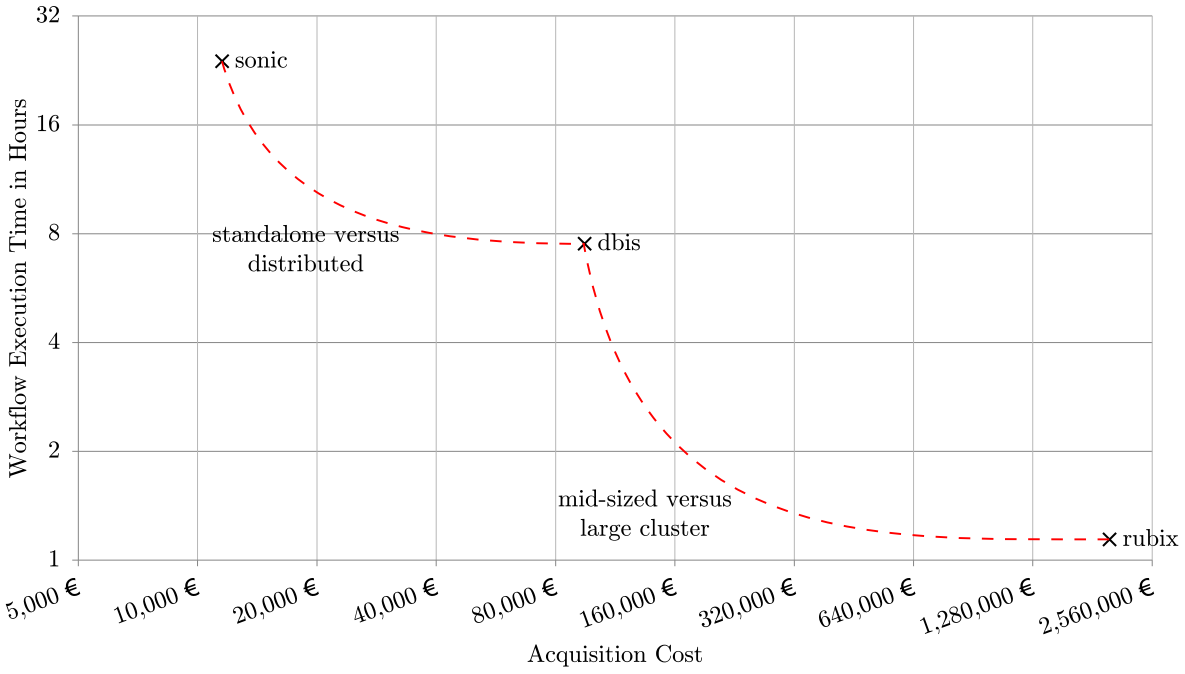


Figure 5.13: Workflow execution time on different hardware configurations (sonic, dbis, rubix) contrasted to hardware acquisition costs. Note that both axes are in logarithmic scale. Evidently, observed throughput does not scale linearly with acquisition cost.

5.3.1 Distributed Scientific Workflow Systems

The scientific workflow management system Pegasus emphasizes scalability, utilizing HTCondor as its underlying execution engine (see Section 2.3.1.1). Pegasus supports a number of static scheduling policies, some of which have been employed in an adaptive context by repeatedly re-computing static schedules (see Section 4.4.2). Pegasus does not allow for iterative workflow structures, since every task has to be explicitly listed in the DAX workflow file. In contrast to Hi-WAY, Pegasus does not provide any means of reproducing scientific experiments across datacenters. Hi-WAY embraces Pegasus by enabling Pegasus workflows to be run on top of Hadoop YARN, as outlined in Section 5.1.2.

Text-based parallel scripting languages like Swift (see Section 2.3.1.2), Snakemake (see Section 2.3.1.3), or Makeflow (Albrecht et al., 2012) are light-weight alternatives to full-fledged workflow management systems. All of these systems provide their own scripting language for the design of data-parallel and iterative workflows. In addition, these systems have in common that they support the scalable execution of implemented workflows on distributed infrastructures. However, they do not implement any means of adaptive scheduling and disregard other features typically present in scientific workflow management systems, such as support for reproducibility.

Nextflow is a novel workflow management system with a strong emphasis on scalability and reproducibility of scientific experiments (Di Tommaso et al., 2017). It brings its own domain-specific language and utilizes operating-system-level virtualization technology in the form of Docker containers. More specifically, it allows workflows or individual tasks to be encapsulated and executed in their own container. Each container has their own isolated user space and containers can easily be shared, which facilitates the design of reproducible workflows, as, for instance, proposed by da Veiga Leprevost et al. (2017). Nextflow enables scalable execution by supporting several general-purpose batch schedulers. Compared to Hi-WAY, execution traces are less detailed and not re-executable. Furthermore, Nextflow does not exploit data-aware and adaptive scheduling.

Toil is a recent multi-language workflow system that supports scalable workflow execution by interfacing with several distributed resource management systems (Vivian et al., 2017). Its supported languages include the Common Workflow Language (CWL) (Amstutz et al., 2016), a YAML-based workflow language that unifies concepts of various other languages, and a custom Python-based DSL that supports the design of iterative workflows. Toil has mostly been developed and used for biomedical analyses, such as for RNA sequencing pipelines (Vivian et al., 2016). Similar to Nextflow, Toil enables sharable and reproducible workflows by allowing tasks to be wrapped in re-usable Docker containers. In contrast to Hi-WAY, Toil does not gather comprehensive provenance and statistics data and, consequently, does not support any means of adaptive workflow scheduling.

The workflow management system Galaxy (see Section 2.3.2.3) neither supports adaptive scheduling nor iterative workflow structures. Similar to Pegasus and as described in Section 5.1.3, Hi-WAY complements Galaxy by allowing exported Galaxy workflows to be run on Hadoop YARN. For a comparative evaluation of Hi-WAY and Galaxy CloudMan, refer to Section 5.2.2.

In contrast to all of the aforementioned workflow management systems, Hi-WAY uses Hadoop as its underlying distributed processing and storage system. While other workflow management systems have provided limited support for outsourcing parts of a workflow to a Hadoop installation (e. g., by specifying an external MapReduce job as a workflow task), Hi-WAY is the first system to natively use Hadoop for handling all of a workflow's tasks and data. Therefore, Hi-WAY does not require an additional processing framework to be installed and maintained if Hadoop is already used in a data center, which, based on the widespread adoption of Hadoop, is not unlikely (see Figure 2.10 on page 23).

5.3.2 Distributed Dataflow Systems

Distributed dataflow systems like Spark (Zaharia et al., 2010) or Flink (Alexandrov et al., 2014) have recently achieved strong momentum both in academia and in the industry. As described in Section 2.2.3.2, these systems operate on semi-structured data and support different programming models, such as SQL-like query languages or real-time stream processing. Departing from the black-box data model along with natively supporting concepts like data streaming and in-memory computing allows these systems to, in many cases, execute sequential processing steps in a data-parallel fashion and circumvent the materialization of intermediate data on the hard disk. It also enables the automatic detection and exploitation of potentials for data parallelism. However, the resulting gains in performance come at the cost of a reduced flexibility and lower level of abstraction for workflow designers, who either (i) have to re-implement existing tools from scratch in a language supported by the utilized dataflow system or (ii) tediously integrate these tools processing unstructured, file-based data, writing a lot of glue code and undermining the advantages of the dataflow system in the progress.

Tez (Saha et al., 2015) is an application master for YARN that enables the execution of DAGs comprising map, reduce, and other tasks. Being a low-level library intended to be interfaced by higher-level applications, external tools consuming and producing file-based data need to be wrapped in order to be used in Tez. For a comparative evaluation between Hi-WAY and Tez, see Section 5.2.1.

Hadoop workflow schedulers like Oozie (Islam et al., 2012) or Azkaban (Sumbaly et al., 2013) have been developed to schedule DAGs consisting of multiple Hadoop jobs (e. g., MapReduce, Pig, or Hive jobs) on a Hadoop installation. In Oozie, tasks composing a workflow are transformed into a number of MapReduce jobs at runtime. When used to run arbitrary scientific workflows, systems like Oozie or Azkaban either (i) introduce unnecessary overhead by wrapping the command-line tasks into degenerate MapReduce jobs or (ii) do not dispatch such tasks to Hadoop, but run them locally instead.

An adaptive scheduler for DAGs of Hadoop jobs has been published by Krish et al. (2015). This scheduler, called ϕSched, tracks the execution time of jobs submitted to Hadoop installations spanning multiple heterogeneous clusters. Based on these execution times and the current load of the underlying clusters, it adaptively determines favorable job-cluster-assignments. In contrast to Hi-WAY, ϕSched operates on the job level, thereby disregarding the fine-grained scheduling capabilities provided by YARN.

Chiron (Ogasawara et al., 2013) is a scalable workflow management system in which data is represented as relations and workflow tasks implement one out of six higher-order functions (e. g., map, reduce, and filter). This departure from the black-box view on data inherent to most scientific workflow management systems enables Chiron to apply concepts of database query optimization to optimize performance through structural workflow reordering (Ogasawara et al., 2011). In contrast to Hi-WAY, Chiron is limited to a single, custom, XML-based workflow language, which does not support iterative workflow structures. Furthermore, while Chiron, like Hi-WAY, is one of few systems in which a workflow's (incomplete) provenance data can be queried during execution of that same workflow, Chiron does not employ this data to perform any adaptive scheduling.

5.4 Summary

In this chapter, we presented Hi-WAY, an application master for executing arbitrary scientific workflows on top of Hadoop YARN. The core features of Hi-WAY are performance gains by means of adaptive and/or data-aware scheduling, scalability through the utilization of Hadoop, a multilingual workflow language interface, support for iterative workflow structures, and tools to facilitate reproducibility of experiments. We described the interface between Hi-WAY and YARN as well as the architecture of Hi-WAY, which is built around the aforementioned concepts. We then outlined a number of experiments, in which real-life workflows from different domains were executed on different computational infrastructures comprising up to 128 worker machines. In the context of these experiments, we also evaluated the ERA scheduler, as presented in Chapter 4, on real cloud infrastructure.

6 Conclusion

We presented solutions for executing data-intensive scientific workflows on shared distributed infrastructures subject to performance variability. We introduced the concepts behind and examples for data-intensive scientific workflows from different domains in Section 2.1. In Section 2.2, we then identified four requirements for the scalable execution of such workflows: (i) a data-parallel and, thus, scalable workflow implementation, (ii) distributed shared-nothing infrastructure such as infrastructure-as-a-service clouds, (iii) a distributed resource manager such as Hadoop YARN that is able to leverage such infrastructure, and (iv) a means of adaptive workflow scheduling that is able to flexibly adjust workflow execution. We outlined how established workflow management systems fail to provide solutions for these requirements in Section 2.3.

To be able to evaluate adaptive workflow scheduling strategies, we surveyed various aspects of performance variability and instability in infrastructure-as-a-service clouds in Section 3.1. In Sections 3.2, 3.3, and 3.4 we presented DynamicCloudSim, a simulation framework that models these aspects of variability. We evaluated DynamicCloudSim's adequacy for simulating cloud infrastructure in Section 3.5. In a subsequent simulation-based comparison of different workflow schedulers in Section 3.6, we noticed substantial performance gains through exploitation of heterogeneity and, in particular, speculative replication of critical tasks.

Based on these findings, in Sections 4.1 and 4.2 we presented ERA, a novel means of integrated task runtime estimation and adaptive workflow scheduling. In Section 4.3, we determined parameter settings for ERA's three heuristics of (i) exploiting heterogeneity in the computational infrastructure, (ii) adapting to dynamic performance changes at runtime, and (iii) replicating pipeline blockers during workflow execution, thereby increasing robustness to instability. In an evaluation in DynamicCloudSim, we found ERA to outperform other established scheduling policies in both the mean and the variance of workflow runtimes when all three of the aforementioned heuristics were combined.

Subsequently, in Section 5.1, we presented Hi-WAY, the first scientific workflow execution engine running on top of the *de facto* industry standard for distributed resource management and storage: Hadoop YARN and HDFS. Hi-WAY's core features are (i) an implementation of ERA alongside other scheduling heuristics, (ii) scalability through utilization of Hadoop, (iii) support for different workflow languages, (iv) the ability to run iterative workflows, and (v) tools to provide reproducibility of experiments. We concluded by performing an in-depth evaluation of these features in Section 5.2. In the context of this evaluation, we found Hi-WAY to scale to cluster sizes of 130 nodes and, likely, beyond. We also confirmed our simulation-based results of ERA outperforming other established scheduling policies, even when historical runtime measurements were scarce.

6.1 Limitations and Future Work

In this section, we discuss the practical applicability of the tools and algorithms introduced in this work, namely DynamicCloudSim, ERA and Hi-WAY. This encompasses shortcomings and limitations as well as an outlook on current developments and desirable future work.

6.1.1 DynamicCloudSim

Arguably, DynamicCloudSim's current models for stragglers and failures during task execution (SAF) are overly simplistic and warrant a revision, as they do not employ any statistical model, but merely assume task runtime to increase by a fixed factor on a straggler machine or in case of failure. Furthermore, DynamicCloudSim's models of variability and instability in the underlying infrastructure are based on performance measurements of Amazon EC2 published between 2008 and 2011. Some of these measurements are probably outdated. In fact, in our experiments on Amazon EC2 presented throughout Sections 3.5, 5.2.1, 5.2.2, and 5.2.3, we did not observe instability or variability to the reported degrees. This observation likely warrants a downward correction of DynamicCloudSim's HET, DCR, and SAF parameters if DynamicCloudSim is employed for simulating the performance of today's EC2 resources.

Note however that other types of distributed shared-nothing infrastructures exist, which are subject to even higher (or substantially lower) degrees of variability. An example would be a cluster shared between multiple users with unlimited access to the cluster's (non-virtualized) resources. Overall, DynamicCloudSim's parameters should always be carefully adjusted to the to-be-simulated infrastructure. We find that the key merit of DynamicCloudSim lies not in its adequacy of mimicking EC2 resources, but rather in its ability to model and adjust different aspects of variability and determine their impact on the performance of scientific workflow schedulers and similar applications.

A difficulty we observed when employing DynamicCloudSim for simulation experiments is that it is not straightforward to quantify and express the resource requirements and capabilities of tasks and compute nodes in discrete values of MIPS or disk I/O and network throughput. Not only do these metrics have to be measured or estimated prior to simulation, but they can also vary over time – a property currently not modeled by DynamicCloudSim. Evidently, this issue is not limited to DynamicCloudSim but, to an even larger degree, is also true for CloudSim, where the performance of a machine can be adjusted only through its MIPS parameter.

Finally, another limitation of CloudSim and, by extension, DynamicCloudSim is their limited scalability, as reported by Depoorter et al. (2008). Since these systems make heavy use of multithreading, they can reach the upper limit of threads supported by a typical Linux kernel when simulating very large infrastructures or user numbers beyond 10,000.

6.1.2 ERA

What truly separates the ERA scheduler from other established scheduling policies such as HEFT or LATE is its adaptation heuristic, i. e., its involvement in determining and maintaining up-to-date runtime measurements for all task-machine-pairs. While in Section 4.3.6 we found this heuristic (employed in combination with ERA's other heuristics) to reduce the variance in workflow runtimes, we did not observe a similar reduction in the mean runtime, despite DynamicCloudSim's arguably high default values for variability and instability (see last section). Evidently, there is a cost associated with occasionally re-evaluating task-machine-assignments based on runtime measurements being outdated rather than favorable. This cost is especially high for workflows comprised mostly of long-running tasks, executed on heterogeneous, yet stable computational infrastructure. Hence, one should carefully consider whether ERA's adaptation heuristic is suitable for a given scenario.

Furthermore, ERA makes a considerable number of assumptions about the to-be-scheduled workload and underlying computational infrastructure. If any of these assumptions are not true in a given scenario, its performance can diminish. For instance, if a machine allows for the execution of more than one task at the same time, ERA might (i) assign many tasks belonging to the same bag to that machine prior to receiving an updated runtime measurement and (ii) by assigning these (similar) tasks to the same machine, unnecessarily impair their performance (and thus deteriorate runtime measurements) as a consequence of co-location and contention for resources. However, we argue that the simplicity of ERA's scheduling heuristics and means of runtime estimation should allow for straightforward extensions and adjustments, enabling a transfer of ERA to other potential use cases.

6.1.3 Hi-WAY

Hi-WAY caters to domain scientists that require a scalable solution for processing large amounts of data using analysis pipelines composed of established command-line tools. Processing black-box data using black-box tasks, Hi-WAY is limited in its scalability by the workflow's degree of parallelism. Hence, Hi-WAY is primarily intended for scientific workflows that have a high associated computational cost and degree of parallelism, as is the case for data-intensive bag-of-tasks workflows.

Emerging from a research project, some of Hi-WAY's components are prototypical and have not been maintained for some time. In particular, this concerns the MySQL and Couchbase adapters of Hi-WAY's Provenance Manager as well as integration of the Galaxy workflow language.

By being able to interpret a number of different workflow languages, Hi-WAY addresses the problem of traditional scientific workflow management systems coupling a custom workflow language to a specific processing framework. Following this rationale, it would make sense for Hi-WAY to also support the Common Workflow Language (Amstutz et al., 2016), which represents a recent effort of the scientific workflow community to establish a *lingua franca* between systems.

A problem we observed was that the reproducibility provided by Hi-WAY in form of Chef recipes that allow for the automated setup of Hadoop, Hi-WAY, and any of its evaluation workflows, can be fragile. The problem here is that this reproducibility depends on the maintenance of several external tools (Chef, Karamel) as well as the availability and constancy of a number of web links (for downloading tools and workflow input data). We have therefore recently experimented with virtualization and containerization support for Hi-WAY by means of Docker containers. Wrapping Hi-WAY installations, workflows, and software dependencies in such interchangeable containers enables a more stable means of reproducibility and is currently being worked on. It would also be another step towards incidental reproducibility, i. e., a form of reproducibility that does not require any additional effort by the workflow designer. In contrast to making a scientific experiment reproducible by specifying all of a workflow's (software and data) dependencies via Chef recipes, ideally the whole execution environment could be exportable in a containerized format that is easy to share and re-enact.

Another recent development of Hi-WAY is an extension of its memory-aware scheduling mechanism. To this end, instead of manually having to assign fixed amounts of memory provided to different (bags of) tasks, memory requirements of tasks are determined automatically and container sizes are tailored accordingly. Similar to how ERA estimates task runtime based on historical runtime measurements, here task memory requirements are estimated based on historical measurements. Since the tasks comprising a given workflow can vary substantially in their memory requirements, this approach should considerably improve memory utilization and, thus, reduce the makespans of memory-bound workflows. Note however that tailoring containers to a workflow's tasks impairs just-in-time scheduling decisions upon container allocation. Hence, just-in-time schedulers like ERA will have to be adjusted and, likely, restricted in their scheduling decisions.

6.2 Outlook

As current technologies mature and evolve across all scientific domains, the amount of generated data will continue to increase beyond what a single machine can feasibly process. While data-parallel programming languages and distributed dataflow systems will find further adoption and dissemination as a consequence of this influx of data, there will always be a need for the scalable execution of analysis pipelines comprising black-box tasks processing black-box data. This will inevitably require scientific workflow management technology to adapt, leading to the following open research topics:

1. The decoupling of conceptual workflow design from underlying execution management, e. g., by agreeing on a uniform computational model for scientific workflows, such as the Common Workflow Language (Amstutz et al., 2016), which can then be interpreted by various distributed processing frameworks.

2. The extension of current workflow management systems with an easy to set-up integration of local, grid, and cloud infrastructures as well as arbitrary compositions thereof (e. g., by means of virtualization and containerization).

3. The continued investigation of (integrated approaches of) task runtime estimation and adaptive scheduling for bag-of-tasks applications in heterogeneous, dynamically changing computational environments, such as shared or composite infrastructures.

4. The development of technologies that facilitate the design of inherently data-parallel scientific workflows without adding additional complexity or barriers to entry for the workflow designer.

5. The provisioning of platforms for sharing workflows alongside their data and executables, making computational scientific experiments truly reproducible with the clicks of a few buttons.

Appendix

Table A: Increasing levels of heterogeneity in the computational infrastructure (HET) and their effects on mean workflow runtimes in minutes. Supplementary data to Figure 4.7 on page 74. *ERA with only its exploitation heuristic enabled and replication and adaptation heuristics disabled.

	RSD parameters for HET				
scheduler	0	0.125	0.25	0.375	0.5
FCFS	1451.56	1453.19	1463.82	1484.63	1471.55
HEFT	1447.26	1454.32	1460.45	1470.63	1444.62
LATE	1519.69	1634.58	1617.89	1602.75	1566.79
ERA*	1450.80	1357.16	1283.59	1239.64	1194.57

Table B: Mean workflow makespan in minutes at increasing levels of dynamic performance changes at runtime in the computational infrastructure (DCR). Supplementary data to Figure 4.8 on page 76.

	RSD parameters for DCR										
α	0	0.125	0.25	0.375	0.5	0.625	0.75	0.875	1	1.125	1.25
0,5	1449	1386	1340	1325	1319	1343	1341	1351	1368	1350	1367
0,2	1449	1372	1317	1304	1319	1345	1366	1366	1381	1386	1387
0,1	1449	1374	1320	1313	1347	1377	1390	1394	1407	1410	1418
0,05	1449	1377	1327	1335	1367	1386	1416	1420	1426	1438	1436
0,02	1450	1382	1338	1357	1389	1426	1444	1456	1459	1455	1468
0,01	1450	1386	1348	1371	1410	1440	1457	1469	1473	1470	1499

Table C: Standard deviations of workflow makespan in minutes at increasing levels of dynamic performance changes at runtime in the computational infrastructure (DCR). Supplementary data to Figure 4.8 on page 76.

	RSD parameters for DCR										
α	0	0.125	0.25	0.375	0.5	0.625	0.75	0.875	1	1.125	1.25
0.5	0.76	18.36	36.93	131.82	92.02	182.81	169.11	181.65	237.53	154.75	209.20
0.2	0.68	17.62	33.23	92.26	114.24	155.97	237.16	187.98	194.92	231.01	189.81
0.1	0.62	16.89	33.29	61.88	142.50	180.60	201.41	190.40	210.29	187.39	191.73
0.05	0.61	16.95	35.19	125.79	116.97	180.49	204.95	201.41	216.44	183.77	192.65
0.02	0.70	16.48	33.82	88.81	122.90	198.03	216.65	194.11	187.15	193.77	202.46
0.01	0.71	16.62	35.14	137.54	143.05	197.23	203.09	206.59	192.19	145.66	255.80

Table D: Increasing levels of dynamic performance changes at runtime in the computational infrastructure (DCR) and their effects on mean workflow runtime using different schedulers. Supplementary data to Figure 4.10 on page 77. *ERA with α set to 0.2 and disabled replication heuristic.

	RSD parameters for DCR				
scheduler	0	0.125	0.25	0.375	0.5
FCFS	1450.38	1450.46	1454.36	1457.70	1439.01
HEFT	1449.02	1547.62	1672.66	1808.06	1894.30
LATE	1653.04	1635.72	1615.09	1590.03	1552.90
ERA*	1449.35	1372.39	1318.42	1292.52	1281.49

Table E: Effects of straggler machines and failures (SAF) on median workflow runtime using ERA in DynamicCloudSim. Supplementary data to Figure 4.11 on page 78.

	SAF parameters for DCR				
ρ	0	0.00625	0.0125	0.01875	0.025
ERA, $\rho = 0$	155.50	211.40	259.54	329.19	473.29
ERA, $\rho = 1$	155.52	176.67	212.93	261.98	305.16
ERA, $\rho = 2$	170.65	191.71	217.62	261.01	305.32

Table F: Effects of straggler virtual machines and failed tasks (SAF) on the standard deviations of workflow runtime using ERA in DynamicCloudSim. Supplementary data to Figure 4.11 on page 78.

	SAF parameters for DCR				
	0	0.00625	0.0125	0.01875	0.025
ERA, $\rho = 0$	13.17	126.97	373.85	411.29	557.98
ERA, $\rho = 1$	12.71	17.23	34.16	42.69	83.87
ERA, $\rho = 2$	13.56	20.13	78.21	67.21	77.28

Table G: Simulated SNV calling workflow runtimes are shown for different schedulers. Supplementary data to Figure 4.12 on page 80.

	FCFS	HEFT	LATE	ERA, $\alpha = 0.5$, $\rho = 0$	ERA, $\alpha = 0.2$, $\rho = 0$	ERA, $\alpha = 0.5$, $\rho = 1$	ERA, $\alpha = 0.2$, $\rho = 1$
Median	187.78	193.72	183.46	186.69	182.17	160.16	162.12
Mean	207.88	206.01	196.38	200.68	198.31	165.40	166.63
STD	121.73	67.53	67.42	77.10	71.53	51.33	27.31

Table H: Summary of the scalability experiment described in Section 5.2.1. The number of provisioned virtual machines is displayed alongside the volume of processed data, average runtime (over three runs), and the incurred cost. *Here, we assume a price of $0.146 per minute, as listed for m3.large instances in EC2's EU West region at the time of writing. We also assume billing per minute and disregard time required to set up the experiment.

number of worker VMs	1	2	4	8	16	32	64	128
number of master VMs	2	2	2	2	2	2	2	2
data volume	8.06 GB	16.97 GB	33.10 GB	69.47 GB	136.14 GB	270.98 GB	546.76 GB	1096.83 GB
avg. runtime in min.	340.12	350.36	351.62	344.82	375.57	372.09	380.24	353.39
runtime std. dev.	1.96	0.14	2.15	1.88	14.84	22.10	22.34	6.01
avg. cost per run*	$2.48	$3.41	$5.13	$8.39	$16.45	$30.78	$61.07	$111.79
avg. cost per GB*	$0.31	$0.20	$0.16	$0.12	$0.12	$0.11	$0.11	$0.10

Bibliography

Abuín, J. M., Pichel, J. C., Pena, T. F., and Amigo, J. (2015). BigBWA: approaching the Burrows-Wheeler aligner to Big Data technologies. *Bioinformatics*, 31(24):4003–4005.

Adhikari, A. N., Peng, J., Wilde, M., Xu, J., Freed, K. F., and Sosnick, T. R. (2012). Modeling large regions in proteins: Applications to loops, termini, and folding. *Protein Science*, 21(1):107–121.

Afgan, E., Baker, D., Coraor, N., Chapman, B., Nekrutenko, A., and Taylor, J. (2010). Galaxy CloudMan: Delivering Cloud Compute Clusters. *BMC Bioinformatics*, 11(Suppl 12):S4.

Afgan, E., Coraor, N., Chilton, J., Baker, D., Taylor, J., and Team", T. G. (2015). Enabling cloud bursting for life sciences within Galaxy. *Concurrency Computation: Practice and Experience*, 27(16):4330–4343.

Albrecht, M., Donnelly, P., Bui, P., and Thain, D. (2012). Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET'12)*.

Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M. J., Schelter, S., Höger, M., Tzoumas, K., and Warneke, D. (2014). The Stratosphere Platform for Big Data Analytics. *VLDB Journal*, 23(6):939–964.

Alkhanak, E. N., Lee, S. P., Rezaei, R., and Parizi, R. M. (2016). Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues. *Journal of Systems and Software*, 113:1–26.

Amstutz, P., Andeer, R., Chapman, B., Chilton, J., Crusoe, M. R., Guimerà, R. V., Hernandez, G. C., Ivkovic, S., Kartashov, A., Leehr, J. K., Ménager, H., Mikheev, M., Pierce, T., Randall, J., Soiland-Reyes, S., Stojanovic, L., and Tijanic, N. (2016). Common Workflow Language, v1.0. *Figshare*.

Angiuoli, S. V., Matalka, M., Gussman, A., Galens, K., Vangala, M., Riley, D. R., Arze, C., White, J. R., White, O., and Fricke, W. F. (2011). CloVR: A virtual machine for automated and portable sequence analysis from the desktop using cloud computing. *BMC Bioinformatics*, 12:356.

Beloglazov, A. and Buyya, R. (2012). Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420.

Benoit, A., Çatalyürek, Ü. V., Robert, Y., and Saule, E. (2013). A Survey of Pipelined Workflow Scheduling: Models and Algorithms. *ACM Computing Surveys*, 45(4):1–36.

Berriman, G. B., Deelman, E., Good, J., Jacob, J., Katz, D. S., Kesselman, C., Laity, A., Prince, T. A., Singh, G., and Su, M.-h. (2004). Montage: A Grid-Enabled Engine for Delivering Custom Science-Grade Mosaics on Demand. *SPIE Conference on Astronomical Telescopes and Instrumentation*, 5493:221–232.

Berthold, M. R., Cebron, N., Dill, F., Di Fatta, G., Gabriel, T. R., Georg, F., Meinl, T., Ohl, P., Sieb, C., and Wiswedel, B. (2006). KNIME: The Konstanz Information Miner. In *Proceedings of the 1st Workshop on Multi-Agent Systems and Simulation (MAS&S'06)*, pages 58–61, Palermo, Italy.

Bessani, A., Brandt, J., Bux, M., Cogo, V., Dimitrova, L., Dowling, J., Gholami, A., Hakimzadeh, K., Hummel, M., Ismail, M., Laure, E., Leser, U., Litton, J. E., Martinez, R., Niazi, S., Reichel, J., and Zimmermann, K. (2015). BiobankCloud: a Platform for the Secure Storage, Sharing, and Processing of Large Biomedical Data Sets. In *First International Workshop on Data Management and Analytics for Medicine and Healthcare (DMAH)*.

Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., and Kennedy, K. (2005). Task Scheduling Strategies for Workflow-based Applications in Grids. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, volume 2, pages 759–767, Cardiff, UK.

Brandt, J., Bux, M., and Leser, U. (2015). Cuneiform: A Functional Language for Large Scale Scientific Data Analysis. *Workshops of the EDBT/ICDT*, 1330:17–26.

Brandt, J., Reisig, W., and Leser, U. (2017). Computation semantics of the functional scientific workflow language Cuneiform. *Journal of Functional Programming*, 27(E22).

Braun, T. D., Siegel, H. J., Beck, N., Boloni, L. L., Maheswarans, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F. (2001). A Comparison Study of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61:810–837.

Brevik, J., Nurmi, D., and Wolski, R. (2006). Predicting Bounds on Queuing Delay for Batch-scheduled Parallel Machines. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization*, pages 213–224.

Bux, M., Brandt, J., Lipka, C., Hakimzadeh, K., Dowling, J., and Leser, U. (2015). Saasfee: Scalable Scientific Workflow Execution Engine. *Proceedings of the 41st International Conference on Very Large Data Bases*, 8(12):1892–1895.

Bux, M., Brandt, J., Witt, C., Dowling, J., and Leser, U. (2017). Hi-WAY: Execution of Scientific Workflows on Hadoop YARN. In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)*, Venice, Italy.

Bux, M. and Leser, U. (2013a). DynamicCloudSim: Simulating Heterogeneity in Computational Clouds. In *Proceedings of the 2nd ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET'13)*, New York, USA.

Bux, M. and Leser, U. (2013b). Parallelization in Scientific Workflow Management Systems. *arXiv preprint, arXiv:1303.7195*.

Bux, M. and Leser, U. (2014). DynamicCloudSim: Simulating Heterogeneity in Computational Clouds. *Future Generation Computer Systems*, 46:85–99.

Buyya, R. and Murshed, M. (2002). GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220.

Cai, Z., Li, X., Ruiz, R., and Li, Q. (2017). A delay-based dynamic scheduling algorithm for bag-of-task workflows with stochastic task execution times in clouds. *Future Generation Computer Systems*, 71:57–72.

Calheiros, R. N., Netto, M. A. S., De Rose, C. A. F., and Buyya, R. (2013). EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of Cloud computing applications. *Software - Practice and Experience*, 43:595–612.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. (2011). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software - Practice and Experience*, 41(1):23–50.

Casavant, T. L. and Kuhl, J. G. (1988). A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154.

Chan, T. F. and Lewis, J. G. (1979). Computing Standard Deviations: Accuracy. *Communications of the ACM*, 22(9):526–531.

Chen, W. and Deelman, E. (2012). WorkflowSim: A Toolkit for Simulating Scientific Workflows in Distributed Environments. In *Proceedings of the 8th IEEE International Conference on eScience*, pages 1–8, Chicago, USA.

Chen, Y., Souaiaia, T., and Chen, T. (2009). PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. *Bioinformatics*, 25(19):2514–2521.

Childs, L. H., Mamlouk, S., Brandt, J., Sers, C., and Leser, U. (2016). SoFIA: A data integration framework for annotating high-throughput datasets. *Bioinformatics*, 32(17):2590–2597.

Cibulskis, K., Lawrence, M. S., Carter, S. L., Sivachenko, A., Jaffe, D., Sougnez, C., Gabriel, S., Meyerson, M., Lander, E. S., and Getz, G. (2013). Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature Biotechnology*, 31(3):213–219.

Cohen-Boulakia, S. and Leser, U. (2011). Search, Adapt, and Reuse: The Future of Scientific Workflows. *SIGMOD Record*, 40(2):6–16.

Cornish, A. and Guda, C. (2015). A Comparison of Variant Calling Pipelines Using Genome in a Bottle as a Reference. *BioMed Research International*, 2015.

Couvares, P., Kosar, T., Roy, A., Weber, J., and Wenger, K. (2007). Workflow Management in Condor. In Taylor, I. J., Deelman, E., Gannon, D., and Shields, M., editors, *Workflows for e-Science*, pages 357–375. Springer, New York, USA, 1st edition.

da Silva, R. F., Filgueira, R., Pietri, I., Jiang, M., Sakellariou, R., and Deelman, E. (2017). A characterization of workflow management systems for extreme-scale applications. *Future Generation Computer Systems*, 75:228–238.

da Silva, R. F., Juve, G., Rynge, M., Deelman, E., and Livny, M. (2015). Online Task Resource Consumption Prediction for Scientific Workflows. *Parallel Processing Letters*, 25(03):1541003.

da Veiga Leprevost, F., Grüning, B. A., Alves Aflitos, S., Röst, H. L., Uszkoreit, J., Barsnes, H., Vaudel, M., Moreno, P., Gatto, L., Weber, J., Bai, M., Jimenez, R. C., Sachsenberg, T., Pfeuffer, J., Vera Alvarez, R., Griss, J., Nesvizhskii, A. I., and Perez-Riverol, Y. (2017). BioContainers: an open-source and community-driven framework for software standardization. *Bioinformatics*, 33(16):2580–2582.

David, M., Dzamba, M., Lister, D., Ilie, L., and Brudno, M. (2011). SHRiMP2: Sensitive yet Practical Short Read Mapping. *Bioinformatics*, 27(7):1011–1012.

Davidson, S. B. and Freire, J. (2008). Provenance and Scientific Workflows: Challenges and Opportunities. *Proceedings of the 2008 ACM SIGMOD Conference*.

de Oliveira, D., Ogasawara, E., Baião, F., and Mattoso, M. (2010). SciCumulus: A Lightweight Cloud Middleware to Explore Many Task Computing Paradigm in Scientific Workflows. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing (CLOUD'10)*, pages 378–385, Miami, USA.

de Oliveira, D., Ogasawara, E., Ocaña, K., Baião, F., and Mattoso, M. (2012). An adaptive parallel execution strategy for cloud-based scientific workflows. *Concurrency and Computation: Practice & Experience*, 24(13):1531–1550.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113.

Decap, D., Reumers, J., Herzeel, C., Costanza, P., and Fostier, J. (2015). Halvade: Scalable sequence analysis with mapReduce. *Bioinformatics*, 31(15):2482–2488.

Deelman, E., Gannon, D., Shields, M., and Taylor, I. (2009). Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540.

Deelman, E., Peterka, T., Altintas, I., Carothers, C. D., Kleese van Dam, K., Moreland, K., Parashar, M., Ramakrishnan, L., Taufer, M., and Vetter, J. (2017). The future of scientific workflows. *The International Journal of High Performance Computing Applications*, page 1094342017704893.

Deelman, E., Singh, G., Livny, M., Berriman, B., and Good, J. (2008). The Cost of Doing Science on the Cloud: The Montage Example. In *Proceedings of the 2008 Conference on Supercomputing (SC'08)*, pages 1–12, Austin, Texas.

Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., and Katz, D. S. (2005). Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237.

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., da Silva, R. F., Livny, M., and Wenger, K. (2015). Pegasus, a Workflow Management System for Large-Scale Science. *Future Generation Computer Systems*, 46:17–35.

Dejun, J., Pierre, G., and Chi, C.-H. (2009). EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications. In *Proceedings of the 7th International Conference on Service Oriented Computing (ICSOC'09)*, pages 197–207, Stockholm, Sweden.

Delimitrou, C. and Kozyrakis, C. (2014). Quasar: Resource-Efficient and QoS-Aware Cluster Management. *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 127–144.

Depoorter, W., Moor, N. D., Vanmechelen, K., and Broeckhove, J. (2008). Scalability of Grid Simulators: An Evaluation. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing (Euro-Par'08)*, pages 544–553, Las Palmas de Gran Canaria, Spain.

Devarakonda, M. V. and Iyer, R. K. (1989). Predictability of Process Resource Usage: A Measurement-Based Study on UNIX. *IEEE Transactions on Software Engineering*, 15(12):1579–1586.

Di Tommaso, P., Chatzou, M., Floden, E. W., Barja, P. P., Palumbo, E., and Notredame, C. (2017). Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319.

Di Tommaso, P., Palumbo, E., Chatzou, M., Prieto, P., Heuer, M. L., and Notredame, C. (2015). The Impact of Docker Containers on the Performance of Genomic Pipelines. *PeerJ*, 3:e1273.

Dinda, P. A. (2002a). A Prediction-based Real-time Scheduling Advisor. In *16th International Parallel and Distributed Processing Symposium (IPDPS)*.

Dinda, P. A. (2002b). Online prediction of the running time of tasks. *Cluster Computing*, 5(3):225–236.

Dobber, M., van der Mei, R., and Koole, G. (2007). A prediction method for job runtimes on shared processors: Survey, statistical analysis and new avenues. *Performance Evaluation*, 64:755–781.

Embrechts, P. and Maejima, M. (2000). An introduction to the theory of self-similar stochastic processes. *International Journal of Modern Physics B*, 14(12n13):1399–1420.

Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud Computing and Grid Computing 360-Degree Compared. In *Proceedings of the 1st Workshop on Grid Computing Environments (GCE'08)*, pages 1–10, Austin, Texas.

Frey, S. and Hasselbring, W. (2011). The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications. *International Journal on Advances in Software*, 4(3 and 4):342–353.

Gao, Y., Rong, H., and Huang, J. Z. (2005). Adaptive grid job scheduling with genetic algorithms. *Future Generation Computer Systems*, 21(1):151–161.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, USA, 1st edition.

Garfinkel, S. L. (2007). An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. Technical report, Technical Report TR-08-07, School for Engineering and Applied Sciences, Harvard University, MA.

Garg, S. K. and Buyya, R. (2011). NetworkCloudSim: Modelling Parallel Applications in Cloud Simulations. In *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC'11)*, pages 105–113, Melbourne, Australia.

Geist, A. and Reed, D. A. (2017). A survey of high-performance computing scaling challenges. *The International Journal of High Performance Computing Applications*, 31(1):104–113.

Gentzsch, W. (2001). Sun Grid Engine: Towards creating a compute power grid. *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 35–36.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5).

Gibbons, R. (1997). A Historical Application Profiler for Use by Parallel Schedulers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 58–77.

Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., and Myers, J. (2007). Examining the Challenges of Scientific Workflows. *IEEE Computer*, 40(12):26–34.

Goble, C. and de Roure, D. (2007). myExperiment: Social Networking for Workflow-Using e-Scientists. *Proceedings of the 2nd Workshop on Workflows in Support of Large-Scale Science (WORKS'07)*.

Goecks, J., Nekrutenko, A., and Taylor, J. (2010). Galaxy: a Comprehensive Approach for Supporting Accessible, Reproducible, and Transparent Computational Research in the Life Sciences. *Genome Biology*, 11(8):R86.

Gordon, M. I., Thies, W., and Amarasinghe, S. (2006). Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, pages 151–162, San Jose, USA.

Herbst, N. R., Huber, N., Kounev, S., and Amrehn, E. (2014). Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. *Concurrency Computation: Practice and Experience*, 26(12):2053–2078.

Hey, T., Tansley, S., and Tolle, K. (2009). *The Fourth Paradigm*. Microsoft Research, Redmond, USA, 2nd edition.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. (2011). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*.

Hirales-Carbajal, A., Tchernykh, A., Röblitz, T., and Yahyapour, R. (2010). A Grid Simulation Framework to Study Advance Scheduling Strategies for Complex Workflow Applications. In *Proceedings of the 24th IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*, pages 1–8, Atlanta, USA.

Hoffa, C., Mehta, G., Freeman, T., Deelman, E., Keahey, K., Berriman, B., and Good, J. (2008). On the Use of Cloud Computing for Scientific Workflows. In *Proceedings of the 4th IEEE International Conference on eScience*, pages 640–645, Indianapolis, USA.

Iosup, A., Yigitbasi, N., and Epema, D. (2011). On the Performance Variability of Production Cloud Services. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'11)*, pages 104–113, Newport Beach, California, USA.

Islam, M., Huang, A. K., Battisha, M., Chiang, M., Srinivasan, S., Peters, C., Neumann, A., and Abdelnur, A. (2012). Oozie: Towards a Scalable Workflow Management System for Hadoop. *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET'12)*.

Iverson, M. A., Özgüner, F., and Potter, L. C. (1999). Statistical Prediction of Task Execution Times through Analytical Benchmarking for Scheduling in a Heterogeneous Environment. In *Eighth Heterogeneous Computing Workshop*, pages 99–111. IEEE Computer Society.

Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J., and Wright, N. J. (2010). Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In *Proceedings of the 2nd International Conference on Cloud Computing Technology and Science (CloudCom'10)*, pages 159–168, Indianapolis, USA.

Juve, G., Chervenak, A., Deelman, E., Bharathi, S., Mehta, G., and Vahi, K. (2012). Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692.

Juve, G. and Deelman, E. (2011). Wrangler: Virtual Cluster Provisioning for the Cloud. In *Proceedings of the 20th International ACM Symposium on High-Performance Parallel and Distributed Computing (HDPC-20)*, pages 277–278, San Jose, USA.

Juve, G., Deelman, E., Vahi, K., Mehta, G., Berriman, B., Berman, B. P., and Maechling, P. (2009). Scientific Workflow Applications on Amazon EC2. In *Proceedings of the 1st Workshop on Cloud-based Services and Applications*, pages 59–66, Oxford, England.

Kahn, S. D. (2011). On the Future of Genomic Data. *Science*, 331(6018):728–729.

Kamthe, A. and Lee, S. Y. (2011). A stochastic approach to estimating earliest start times of nodes for scheduling DAGs on heterogeneous distributed computing systems. *Cluster Computing*, 14(4):377–395.

Kim, D., Pertea, G., Trapnell, C., Pimentel, H., Kelley, R., and Salzberg, S. L. (2013). TopHat2: Accurate Alignment of Transcriptomes in the Presence of Insertions, Deletions and Gene Fusions. *Genome Biology*, 14:R36.

Koboldt, D. C., Chen, K., Wylie, T., Larson, D. E., McLellan, M. D., Mardis, E. R., Weinstock, G. M., Wilson, R. K., and Ding, L. (2009). VarScan: Variant Detection in Massively Parallel Sequencing of Individual and Pooled Samples. *Bioinformatics*, 25(17):2283–2285.

Köster, J. and Rahmann, S. (2012). Snakemake – a Scalable Bioinformatics Workflow Engine. *Bioinformatics*, 28(19):2520–2522.

Krish, K. R., Anwar, A., and Butt, A. R. (2015). PhiSched: A Heterogeneity-Aware Hadoop Workflow Scheduler. In *Proceedings of the 22nd IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 255–264.

Kumar, V. S. A., Marathe, M. V., Parthasarathy, S., and Srinivasan, A. (2009). Scheduling on unrelated machines under tree-like precedence constraints. *Algorithmica (New York)*, 55(1):205–226.

Langmead, B. and Salzberg, S. (2012). Fast Gapped-Read Alignment with Bowtie 2. *Nature Methods*, 9:357–359.

Langmead, B., Schatz, M. C., Lin, J., Pop, M., and Salzberg, S. L. (2009a). Searching for SNPs with cloud computing. *Genome Biology*, 10(11):R134.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009b). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25.

Lee, K., Paton, N. W., Sakellariou, R., Deelman, E., Fernandes, A. A. A., and Mehta, G. (2009). Adaptive Workflow Processing and Execution in Pegasus. *Concurrency and Computation: Practice and Experience*, 21(16):1965–1981.

Lenstra, J. K., Shmoys, D. B., and Tardos, É. (1990). Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1-3):259–271.

Li, B. and Leal, S. M. (2008). Methods for Detecting Associations with Rare Variants for Common Diseases: Application to Analysis of Sequence Data. *American Journal of Human Genetics*, 83(3):311–321.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., and Durbin, R. (2009). The Sequence Alignment/Map Format and SAMtools. *Bioinformatics*, 25(16):2078–2079.

Li, H. and Homer, N. (2010). A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483.

Liew, C. S., Atkinson, M. P., Galea, M., Ang, T. F., Martin, P., and Van Hemert, J. I. (2017). Scientific Workflows: Moving Across Paradigms. *ACM Computing Surveys*, 49(4):66.

Lipka, C. (2014). *Implementierung eines NGS-Workflows unter Verwendung von MapReduce und Hadoop*. Bachelor's thesis, Humboldt-Universität zu Berlin.

Litzkow, M. J., Livny, M., and Mutka, M. W. (1988). Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS'88)*, pages 104–111, San Jose, USA.

Liu, J., Pacitti, E., Valduriez, P., and Mattoso, M. (2015). A Survey of Data-Intensive Scientific Workflow Management. *Journal of Grid Computing*, 13(4):457–493.

Liu, X., Ni, Z., Yuan, D., Jiang, Y., Wu, Z., Chen, J., and Yang, Y. (2011). A novel statistical time-series pattern based interval forecasting strategy for activity durations in workflow systems. *The Journal of Systems and Software*, 84:354–376.

Lloyd, W., Pallickara, S., David, O., Arabi, M., and Rojas, K. (2017). Mitigating Resource Contention and Heterogeneity in Public Clouds for Scientific Modeling Services. In *Proceedings of the IEEE International Conference on Cloud Engineering*, pages 159–166.

Maji, A. K., Mitra, S., Zhou, B., Bagchi, S., and Verma, A. (2014). Mitigating Interference in Cloud Services by Middleware Reconfiguration. In *Proceedings of the 15th International Middleware Conference*, pages 277–288.

Malawski, M., Juve, G., Deelman, E., and Nabrzyski, J. (2015). Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. *Future Generation Computer Systems*, 48:1–18.

Maleki-Dizaji, S., Rolfe, M., Fisher, P., and Holcombe, M. (2009). A Systematic Approach to Understanding Bacterial Responses to Oxygen Using Taverna and Webservices. In *Proceedings of the 13th International Conference on Biomedical Engineering (ICBME'08)*, pages 77–80, Singapore.

Mandal, A., Kennedy, K., Koelbel, C., Marin, G., Mellor-Crummey, J., Liu, B., and Johnsson, L. (2005). Scheduling Strategies for Mapping Application Workflows onto the Grid. In *Proceedings on the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, pages 125–134, Durham, USA.

Massie, M., Nothaft, F., Hartl, C., Kozanitis, C., Schumacher, A., Joseph, A. D., and Patterson, D. A. (2013). ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing. Technical report, UCB/EECS-2013-207, Electrical Engineering and Computer Sciences, University of California at Berkeley.

Mayer, B., Worley, P. H., da Silva, R. F., and Gaddis, A. L. (2015). Climate Science Performance, Data and Productivity on Titan. In *Cray User Group Conference*.

McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M., and DePristo, M. A. (2010). The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303.

Mell, P. and Grance, T. (2009). The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology.

Merdan, M., Moser, T., Wahyudin, D., Biffl, S., and Vrba, P. (2008). Simulation of workflow scheduling strategies using the MAST test management system. In *Proceedings of the 10th International Conference on Control, Automation, Robotics and Vision (ICARCV'08)*, pages 1172–1177, Hanoi, Vietnam.

Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., and Goble, C. (2010). Taverna, reloaded. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM'10)*, pages 471–481, Heidelberg, Germany.

Momcheva, I. and Tollerud, E. (2015). Software Use in Astronomy: An Informal Survey. *arXiv preprint, arXiv:1507.03989*.

Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. (2011). CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI 2011)*.

Nekrutenko, A. and Taylor, J. (2012). Next-generation sequencing data interpretation: enhancing reproducibility and accessibility. *Nature Reviews Genetics*, 13(9):667–672.

Nordberg, H., Bhatia, K., Wang, K., and Wang, Z. (2013). BioPig: A Hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014–3019.

Nothaft, F. A., Massie, M., Danford, T., Zhang, Z., Laserson, U., Yeksigian, C., Kottalam, J., Ahuja, A., Hammerbacher, J., Linderman, M., Franklin, M. J., Joseph, A. D., and Patterson, D. A. (2015). Rethinking Data-Intensive Science Using Scalable Analytics Systems. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 631–646.

Novaković, D., Vasić, N., Novaković, S., Kostić, D., and Bianchini, R. (2013). Deepdive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceeding of the 2013 USENIX Annual Technical Conference*, pages 219–230.

Ogasawara, E., Dias, J., de Oliveira, D., Porto, F., Valduriez, P., and Mattoso, M. (2011). An Algebraic Approach for Data-Centric Scientific Workflows. In *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB'11)*, volume 4, pages 1328–1339, Seattle, USA.

Ogasawara, E., Dias, J., Silva, V., Chirigati, F., de Oliveira, D., Porto, F., Valduriez, P., and Mattoso, M. (2013). Chiron: A Parallel Engine for Algebraic Scientific Workflows. *Concurrency and Computation: Practice and Experience*, 25(16):2327–2341.

Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M. R., Wipat, A., and Li, P. (2004). Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054.

Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD Conference*, pages 1099–1110, Vancouver, Canada.

Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. (2008). An Early Performance Analysis of Cloud Computing Services for Scientific Computing. Technical report, TU Delft.

Ostermann, S., Plankensteiner, K., Prodan, R., and Fahringer, T. (2010). GroudSim: An Event-Based Simulation Framework for Computational Grids and Clouds. In *CoreGRID/ERCIM Workshop on Grids, Clouds and P2P Computing in conjunction with EuroPAR 2010*, pages 305–313, Ischia, Italy.

Pabinger, S., Dander, A., Fischer, M., Snajder, R., Sperk, M., Efremova, M., Krabichler, B., Speicher, M. R., Zschocke, J., and Trajanoski, Z. (2014). A Survey of Tools for Variant Analysis of Next-Generation Genome Sequencing Data. *Briefings in Bioinformatics*, 15(2):256–278.

Palankar, M., Iamnitchi, A., Ripeanu, M., and Garfinkel, S. (2008). Amazon S3 for Science Grids: a Viable Solution? In *Proceedings of the 1st Workshop on Data-aware Distributed Computing (DADC'08)*, pages 55–64, Boston, USA.

Pelletingeas, C. (2010). *Performance Evaluation of Virtualization with Cloud Computing*. Master's thesis, Edinburgh Napier University.

Pennisi, E. (2011). Will Computers Crash Genomics? *Science*, 331(6018):666–668.

Pireddu, L., Leo, S., and Zanetti, G. (2011). Seal: A distributed short read mapping and duplicate removal tool. *Bioinformatics*, 27(15):2159–2160.

Prins, P., de Ligt, J., Tarasov, A., Jansen, R. C., Cuppen, E., and Bourne, P. E. (2015). Toward effective software solutions for big biology. *Nature Biotechnology*, 33(7):686–687.

Prodan, R. and Fahringer, T. (2005). Dynamic Scheduling of Scientific Workflow Applications on the Grid: A Case Study. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 687–694, Santa Fe, USA.

Rahman, M., Hassan, R., Ranjan, R., and Buyya, R. (2013). Adaptive Workflow Scheduling for Dynamic Grid and Cloud Computing Environment. *Concurrency Computation Practice and Experience*, 25:1816–1842.

Ramírez-Alcaraz, J. M., Tchernykh, A., Yahyapour, R., Schwiegelshohn, U., Quezada-Pina, A., González-García, J. L., and Hirales-Carbajal, A. (2011). Job Allocation Strategies with User Run Time Estimates for Online Scheduling in Hierarchical Grids. *Journal of Grid Computing*, 9:95–116.

Reuther, A., Byun, C., Arcand, W., Bestor, D., Bergeron, B., Hubbell, M., Jones, M., Michaleas, P., Prout, A., Rosa, A., and Kepner, J. (2016). Scheduler Technologies in Support of High Performance Data Analysis. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference*.

Rheinländer, A., Leser, U., and Graefe, G. (2017). Optimization of Complex Data Flows with User-Defined Functions. *ACM Computing Surveys*, 50(3):article 38.

Ross, S. M. (2014). Variations on Brownian Motion. In *Introduction to Probability Models*, pages 612–614. Elsevier, 11th edition.

Sadhasivam, S., Nagaveni, N., Jayarani, R., and Ram, R. V. (2009). Design and Implementation of an Efficient Two-level Scheduler for Cloud Computing Environment. In *Proceedings of the 2009 International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom'09)*, pages 884–886, Kottayam, India.

Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A., and Curino, C. (2015). Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. *Proceedings of the 2015 ACM SIGMOD Conference*.

Santana-Perez, I., Ferreira da Silva, R., Rynge, M., Deelman, E., Pérez-Henández, M. S., and Corcho, O. (2014). Leveraging Semantics to Improve Reproducibility in Scientific Workflows. *Reproducibility at XSEDE Workshop*.

Saubern, S., Guha, R., and Baell, J. B. (2011). KNIME Workflow to Assess PAINS Filters in SMARTS Dormat. Comparison of RDKit and Indigo Cheminformatics libraries. *Molecular Informatics*, 30(10):847–850.

Schad, J., Dittrich, J., and Quiané-Ruiz, J.-A. (2010). Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proceedings of the VLDB Endowment*, 3(1):460–471.

Schatz, M. C. (2009). CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369.

Schmied, C., Steinbach, P., Pietzsch, T., Preibisch, S., and Tomancak, P. (2016). An automated workflow for parallel processing of large multiview SPIM recordings. *Bioinformatics*, 32(7):1112–1114.

Schoenherr, S., Forer, L., Weissensteiner, H., Specht, G., Kronenberg, F., and Kloss-Brandstaetter, A. (2012). Cloudgene: A graphical execution platform for MapReduce programs on private and public clouds. *BMC Bioinformatics*, 13:200.

Schroeder, B. and Gibson, G. A. (2006). A large-scale study of failures in high-performance-computing systems. In *Proceedings of the 36th International Conference on Dependable Systems and Networks (DSN'06)*, pages 249–258, Philadelphia, USA.

Schuh, H. (2015). *HiwayDB: Storing Provenance Traces of Scientific Workflows Executed on Hadoop*. Diploma thesis, Humboldt-Universität zu Berlin.

Schumacher, A., Pireddu, L., Niemenmaa, M., Kallio, A., Korpelainen, E., Zanetti, G., and Heljanko, K. (2014). SeqPig: Simple and scalable scripting for large sequencing data sets in hadoop. *Bioinformatics*, 30(1):119–120.

Shendure, J. and Ji, H. (2008). Next-generation DNA sequencing. *Nature Biotechnology*, 26(10):1135–1145.

Sherry, S. T., Ward, M. H., Kholodov, M., Baker, J., Phan, L., Smigielski, E. M., and Sirotkin, K. (2001). dbSNP: the NCBI database of genetic variation. *Nucleic Acids Research*, 29(1):308–311.

Sieb, C., Meinl, T., and Berthold, M. R. (2007). Parallel and Distributed Data Pipelining with KNIME. *The Mediterranean Journal of Computers and Networks*, 3(2):43–51.

Sonmez, O., Yigitbasi, N., Iosup, A., and Epema, D. (2009). Trace-Based Evaluation of Job Runtime and Queue Wait Time Predictions in Grids. In *18th ACM International Symposium on High Performance Distributed Computing*, pages 111–120, Munich, Germany.

Staples, G. (2006). TORQUE Resource Manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Tampa, Florida.

Stein, L. D. (2010). The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207.

Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., and Robinson, G. E. (2015). Big Data: Astronomical or Genomical? *PLoS Biology*, 13(7):e1002195.

Sumbaly, R., Kreps, J., and Shah, S. (2013). The Big Data Ecosystem at LinkedIn. *Proceedings of the 2013 ACM SIGMOD Conference*.

Tang, X., Li, K., Liao, G., Fang, K., and Wu, F. (2011). A stochastic scheduling algorithm for precedence constrained tasks on Grid. *Future Generation Computer Systems*, 27(8):1083–1091.

Taylor, I. J., Deelman, E., Gannon, D. B., and Shields, M. (2007). *Workflows for e-Science: Scientific Workflows for Grids*. Springer.

Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356.

The 1000 Genomes Project Consortium (2015). A Global Reference for Human Genetic Variation. *Nature*, 526(7571):68–74.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629.

Topcuoglu, H., Hariri, S., and Wu, M.-Y. (2002). Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

Trapnell, C., Roberts, A., Goff, L., Pertea, G., Kim, D., Kelley, D. R., Pimentel, H., Salzberg, S. L., Rinn, J. L., and Pachter, L. (2012). Differential Gene and Transcript Expression Analysis of RNA-seq Experiments with TopHat and Cufflinks. *Nature Protocols*, 7(3):562–578.

Tsafrir, D., Etsion, Y., and Feitelson, D. (2007). Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803.

Ullman, J. D. (1975). NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10(3):384–393.

Van Dijk, E. L., Auger, H., Jaszczyszyn, Y., and Thermes, C. (2014). Ten Years of Next-Generation Sequencing Technology. *Trends in Genetics*, 30(9):418–426.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet Another Resource Negotiator. *Proceedings of the Fourth ACM Symposium on Cloud Computing (SOCC'13)*.

Vivian, J., Rao, A., Nothaft, F. A., Ketchum, C., Armstrong, J., Novak, A., Pfeil, J., Narkizian, J., Deran, A. D., Musselman-Brown, A., Schmidt, H., Amstutz, P., Craft, B., Goldman, M., Rosenbloom, K., Cline, M., O'Connor, B., Hanna, M., Birger, C., Kent, W. J., Patterson, D. A., Joseph, A. D., Zhu, J., Zaranek, S., Getz, G., Haussler, D., and Paten, B. (2016). Rapid and Efficient Analysis of 20,000 RNA-seq Samples with Toil. *bioRxiv*, 062497.

Vivian, J., Rao, A. A., Nothaft, F. A., Ketchum, C., Armstrong, J., Novak, A., Pfeil, J., Narkizian, J., Deran, A. D., Musselman-Brown, A., Schmidt, H., Amstutz, P., Craft, B., Goldman, M., Rosenbloom, K., Cline, M., O'Connor, B., Hanna, M., Birger, C., Kent, W. J., Patterson, D. A., Joseph, A. D., Zhu, J., Zaranek, S., Getz, G., Haussler, D., and Paten, B. (2017). Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, 35(4):314–316.

Wandelt, S., Rheinländer, A., Bux, M., Thalheim, L., Haldemann, B., and Leser, U. (2012). Data Management Challenges in Next Generation Sequencing. *Datenbank-Spektrum*, 12(3):161–171.

Wang, D. (2017). hppRNA - a Snakemake-based handy parameter-free pipeline for RNA-Seq analysis of numerous samples. *Briefings in Bioinformatics*, bbw143.

Wang, K., Li, M., and Hakonarson, H. (2010). ANNOVAR: Functional Annotation of Genetic Variants from High-Throughput Sequencing Data. *Nucleic Acids Research*, 38(16):e164.

Wang, Y., Mehta, G., Mayani, R., Lu, J., Souaiaia, T., Chen, Y., Clark, A., Yoon, H. J., Wan, L., Evgrafov, O. V., Knowles, J. A., Deelman, E., and Chen, T. (2011). RseqFlow: workflows for RNA-Seq data analysis. *Bioinformatics*, 27(18):2598–2600.

White, T. (2012). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, USA, 3rd edition.

Wiewiórka, M. S., Messina, A., Pacholewska, A., Maffioletti, S., Gawrysiak, P., and Okoniewski, M. J. (2014). SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, 30(18):2652–2653.

Wilde, M., Hategan, M., Wozniak, J. M., Clifford, B., Katz, D. S., and Foster, I. (2011). Swift: A Language for Distributed Parallel Scripting. *Parallel Computing*, 37(9):633–652.

Woitaszek, M., Dennis, J. M., and Sines, T. R. (2011). Parallel High-resolution Climate Data Analysis using Swift. In *Proceedings of the 4th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS'11)*, pages 5–14, Seattle, USA.

Wolfien, M., Rimmbach, C., Schmitz, U., Jung, J. J., Krebs, S., Steinhoff, G., David, R., and Wolkenhauer, O. (2016). TRAPLINE: A Standardized and Automated Pipeline for RNA Sequencing Data Analysis, Evaluation and Annotation. *BMC Bioinformatics*, 17:21.

Wolski, R. (1998). Dynamically forecasting network performance using the Network Weather Service. *Cluster Computing*, 1(1):119–132.

Wolski, R., Spring, N., and Hayes, J. (2000). Predicting the CPU Availability of Time-Shared Unix Systems on the Computational Grid. *Cluster Computing*, 3(4):293–301.

Wolski, R., Spring, N. T., and Hayes, J. (1999). Network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5):757–768.

Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., Bhagat, J., Belhajjame, K., Bacall, F., Hardisty, A., Nieva de la Hidalga, A., Balcazar Vargas, M. P., Sufi, S., and Goble, C. (2013). The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud. *Nucleic Acids Research*, 41:557–561.

Wu, D., Zhu, L., Xu, X., Sakr, S., Sun, D., and Lu, Q. (2016). Building Pipelines for Heterogeneous Execution Environments for Big Data Processing. *IEEE Software*, 33(2):60–67.

Wu, L., Garg, S. K., and Buyya, R. (2011). SLA-Based Resource Allocation for Software as a Service Provider (SaaS) in Cloud Computing Environments. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'11)*, pages 195–204, Newport Beach, California, USA.

Wu, Y., Hwang, K., Yuan, Y., and Zheng, W. (2010). Adaptive Workload Prediction of Grid Performance in Confidence Windows. *IEEE Transactions on Parallel and Distributed Systems*, 21(7):925–938.

Yang, L., Foster, I., and Schopf, J. M. (2003a). Homeostatic and Tendency-based CPU Load Predictions. In *17th International Parallel and Distributed Processing Symposium (IPDPS)*.

Yang, L., Schopf, J. M., and Foster, I. (2003b). Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments. In *ACM/IEEE Conference on Supercomputing*, Phoenix, USA.

Yoo, A. B., Jette, M. A., and Grondona, M. (2003). SLURM: Simple Linux Utility for Resource Management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60.

Yu, J. and Buyya, R. (2005). A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3:171–200.

Yu, J. and Buyya, R. (2006). A Budget Constrained Scheduling of Workflow Applications on Utility Grids using Genetic Algorithms. In *Proceedings of the 1st Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, pages 1–10, Paris, France.

Yu, J., Buyya, R., and Ramamohanarao, K. (2008). Workflow scheduling algorithms for grid computing. In *Metaheuristics for scheduling in distributed computing environments*, volume 146, pages 173–214. Springer.

Yu, Z. and Shi, W. (2007). An Adaptive Rescheduling Strategy for Grid Workflow Applications. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium (IPDPS'07)*, pages 1–8, Los Angeles, USA.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster Computing with Working Sets. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10)*.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. (2008). Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 29–42, San Diego, USA.

Zhang, Y., Sun, W., and Inoguchi, Y. (2008). Predict task running time in grid environments based on CPU load predictions. *Future Generation Computer Systems*, 24:489–497.

Zhang, Z., Li, Z., Wu, K., Li, D., Li, H., Peng, Y., and Lu, X. (2014). VMThunder: Fast Provisioning of Large-Scale Virtual Machine Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3328–3338.

Zhao, Y., Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Raicu, I., Stef-Praun, T., and Wilde, M. (2007). Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *Proceedings of the 1st IEEE International Workshop on Scientific Workflows (SWF'07)*, pages 199–206, Salt Lake City, USA.

Zhao, Y., Li, Y., Raicu, I., Lu, S., Tian, W., and Liu, H. (2015). Enabling scalable scientific workflow management in the Cloud. *Future Generation Computer Systems*, 46:3–16.

Zhou, X., Krabbenhöft, H., Niinimäki, M., Depeursinge, A., Möller, S., and Müller, H. (2009). An Easy Setup for Parallel Medical Image Processing: Using Taverna and ARC. *Studies in Health Technology and Informatics*, 147:41–50.

Zou, Q., Li, X.-B., Jiang, W.-R., Lin, Z.-Y., Li, G.-L., and Chen, K. (2013). Survey of MapReduce frame operation in bioinformatics. *Briefings in Bioinformatics*, 15(4):637–647.

Abbrevations

AM	**A**pplication **M**aster for YARN	21
DAX	Pegasus' workflow language **DA**G in **X**ML	29
DCR	**D**ynamic performance **C**hanges at **R**untime of computational resources, an aspect of performance variability modeled by DynamicCloudSim	43
DN	HDFS's **D**ata **N**ode	21
EC2	Amazon's infrastructure-as-a-service cloud, **E**lastic **C**ompute **C**loud	17
ERA	An adaptive scheduling policy based on **E**xploitation, **R**eplication, and **A**daptation	65
FCFS	**F**irst-**C**ome-**F**irst-**S**erved scheduling	24
HDFS	**H**adoop's **D**istributed **F**ile **S**ystem	21
HEFT	**H**eterogeneous **E**arliest **F**inish **T**ime	25
HET	**Het**erogeneity between computational resources, an aspect of performance variability modeled by DynamicCloudSim	42
Hi-WAY	**Hi**-WAY **W**orkflow **A**pplication Master for **Y**ARN	91
LATE	**L**ongest **A**pproximate **T**ime to **E**nd	26
NM	YARN's **N**ode **M**anager	21
NN	HDFS's **N**ame **N**ode	21
RSD	**R**elative **S**tandard **D**eviation, a standardized measure of variation	42
RM	YARN's **R**esource **M**anager	21
SAASFEE	A **sc**a**l**a**b**le **s**cientific work**f**low **e**xecution **e**ngine that bundles the Cuneiform workflow language and Hi-WAY	90
SAF	**S**traggler **A**nd otherwise **F**aulty computational resources, an aspect of performance variability modeled by DynamicCloudSim	44
SNV	**S**ingle **N**ucleotide **V**ariant	11
YARN	Hadoop's resource management unit, **Y**et **A**nother **R**esource **N**egotiator	21

Index

List of Figures

List of Tables

Selbständigkeitserklärung

Ich erkläre, dass ich die Dissertation selbständig und nur unter Verwendung der von mir gemäß § 7 Abs. 3 der Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 126/2014 am 18.11.2014, angegebenen Hilfsmittel angefertigt habe.

Berlin, den 28. Februar 2018 Marc Nicolas Bux